Game-Changing Advances in Windows Shellcode Analysis

DEFCON 31, 2023

Bramwell Brizendine[1], Jacob Hince[2], Austin Babcock[2], Max Kersten[3]
[1] University of Alabama in Huntsville, Huntsville, AL, USA
[2] Dakota State University in Madison, SD, USA
[3] Trellix

bramwell.brizendine@uah.edu

Abstract. SHAREM is a groundbreaking shellcode analysis and emulation framework, with several unprecedented, novel features. Not only can SHAREM emulate more than 20,000 WinAPIs and virtually all Windows syscalls, but it also has a highly accurate shellcode disassembler. This disassembly uniquely can integrate emulation data, to produce virtually flawless disassembly. With encoded shellcode samples, SHAREM can not only emulate them, revealing WinAPIs utilized, but it can also recover the original form of the shellcode, even if efforts were made to prevent this. By default, SHAREM will generate the disassembly for the decoded form of a shellcode, rather than its encoded form.

Keywords: malware analysis, malware detection, shellcode analysis, emulation, SHAREM, disassembler

1 Introduction

Shellcode is an omnipresent element of the malware ecosystem, as much malware makes use of shellcode as a means to obscure the malware author's intentions and complicate analysis. Additionally, shellcode is heavily used in software exploitation, to achieve malicious functionality in a position-independent fashion. When shellcode is injected into a process, it is unable to directly call any functions through normal means, so it must engage in introspection to discover needed runtime addresses of API functions, allowing them to be called. Shellcode often does this in a very indirect fashion, possibly even generating checksums that are then compared to pre-computed checksums, rather than directly calling the desired APIs. Without a framework such as SHAREM, there would be no possible way to even know which APIs were being called. The traditional approach to such a predicament is to debug the shellcode. This could be done utilizing a debugger, such as WinDbg or x32Dbg; it is tedious and time-consuming compared to SHAREM, particularly if advanced obfuscation is in place.

The very premise by which modern Windows shellcode operates is a heavy reliance on introspection, in order to discover otherwise clandestine API addresses. It all begins by a technique called walking the Process Environment Block (PEB), which can be used to discover various internal features of the Windows operating system, not limited to the base address of modules, such as NTDLL.dll or kernel32.dll. Once this base address is found, then we have discovered the start of a PE file. Then we can then apply the various well-documented structures of the PE file, until at length we are able to discover the exports directory. Then, we can eventually discover runtime addresses for desired functions. Clever shellcode will be highly modular, with various helper functions to help facilitate different stages of traversing the PE file format, such that numerous runtime addresses for different APIs could easily be determined and then saved to memory, allowing them to easily be called upon later on.

With the advent of ShellWasp [1] to facilitate shellcode comprised entirely or almost exclusively with Windows syscalls, a whole new approach to shellcode has arisen. We can call this syscall shellcode, and with this variety of shellcode, it is not even necessary to traverse the exports directory or obtain runtime addresses for functions, as we can call Windows syscalls directly. Still, an entirely different form of introspection is required, as we must again traverse the PEB, but this time we look for different offsets, to find the OSMajorVersion and OSBuild. With syscall shellcode, powerful malicious functionality can be achieved, and even individuals who are well acquainted with the usage of WinAPI functions in shellcode, may be confused at Windows syscalls in shellcode.

https://t.me/learningnets

While there are numerous variations of tools designed to both detect and contain malware, a framework exclusively devoted to shellcode is a hitherto unmet need, of great interest both to the malware analyst as well as exploit developers, or for others engaged in offensive security. Understanding Windows shellcode can be a challenging task, particularly if the author wishes to obscure what they are doing. Not only can they use checksums in place of plaintext names for different WinAPI functions, but they can also use complicated encoding schemes to make deciphering what they have done more challenging. Prior to the start of SHAREM, there was only one currently available tool capable of enumerating API's in shellcode, Scdbg [2]. While Scdbg is highly effective within the confines of its limited functionality, it should be noted that it only supports 242 APIs. After being awarded an NCAE/NSA research grant for SHAREM, another tool, SpeakEasy [3], was released, which also has the ability to enumerate WinAPIs. To ensure we were not influenced by SpeakEasy during the two-year period of performance for the grant, we have avoided looking at its functionality, aside from cursory usage to test some shellcode samples. We note that SpeakEasy's primary purpose was not to analyze shellcode, and SHAREM has vastly more features pertaining to shellcode. We also note that some shellcode samples that failed with Scdbg or SpeakEasy, were able to be successfully deobfuscated with SHAREM. Thus, there was an unmet demand for a shellcode analysis framework capable of enumerating more than 10,000 WinAPIs, as it can be entirely unpredictable which WinAPIs malicious actors may employ. Furthermore, there had never been any tool capable of emulating and enumerating Windows syscalls. Additionally, it is important to point out that there are many functions that will not use mere parameters. One function could have only one parameter, and that single parameter could be a pointer to a structure whose members could consist of multiple other structures as well as other members. How would an existing tool capable of emulating functions handle this? Would it simply display the pointer to the parent structure? That would be of minimal value, and therefore a shellcode emulation framework capable of applying structures to function parameters was desperately needed. Being able to take a shellcode and enumerate all existing WinAPI functions and Windows syscalls with a high degree of accuracy, while generating a highly detailed, comprehensive report that also includes virtually flawless disassembly was an unmet need.

Disassembly of modern Windows shellcode in leading disassemblers, such as IDA Pro, Ghidra, or Binary Ninja, could lead to disastrous results, some of which are comically erroneous. If we think about a disassembler, its primary purpose is to be able to accurately distinguish between data and instructions, or presenting those correctly in a way that is easily readable and accessible to a human analyst. If the distinction between data and instructions is fully accurate at every byte and at every offset, then very likely, then the resulting disassembly will have a high degree of accuracy. With well-formed and non-malicious PE files, this can be a straightforward process. However, modern Windows shellcode can be highly unorthodox, with data such as strings or checksums interspersed at seemingly random locations throughout the shellcode. The shellcode is often highly compact, with many jump conditionals and function calls, while not necessarily paying attention to traditional conventions. Ultimately the problem is that different data is misinterpreted as instructions, while at the same time the disassembly of some instructions may begin at an incorrect offset, resulting in very misleading disassembly. For whatever reason, highly inaccurate disassembly seems to be much more frequently associated with shellcode.

Many shellcode samples employ advanced obfuscation methods that may require the use of a debugger to analyze them, as a disassembler would only reveal a decoder stub and then encoded shellcode. Having to step through a debugger to discover the decoded shellcode would be tedious. Thus, there is a need to be able to take an encoded shellcode and allow the disassembler to disassemble its decoded form, and to do so in an automated fashion with minimal or no effort. It is an idea that seems incomprehensible to some, as it combines static analysis with dynamic analysis in a way that has never been done before, and it does so seamlessly without fanfare.

Shellcode has been a popular part of malware and exploitation for decades. Many people rely upon tools to generate variations on prebuilt shellcode, while applying an encoding scheme. Some may go online and download an existing shellcode, perhaps making some slight modifications, although many may not be able to modify shellcode without special training or research on their own. In summary, existing approaches to the analysis of shellcode have been lacking, and they can necessitate time-consuming and tedious actions to unravel actions are being performed within the shellcode, often demanding some level of expertise. While there are some tools that can identify a limited subset of WinAPI functionality, often the analysis may be incomplete, due to errors or lack of support for needed APIs; additionally, more advanced shellcode may stymie their efforts. Still, a dedicated researcher could make use of a more dynamic approach, such as placing the shellcode in a special harness program, allowing it to be analyzed with a debugger. While that is one option, it is important to bear in mind that many shellcode samples may prematurely terminate if certain actions are unsuccessful. Thus, debugging may not be an option, unless one were to manually change certain values during debugging to simulate success.

https://t.me/learningnets

1.1 Contribution

In this paper, we present a novel framework for shellcode analysis called SHAREM, or the Shellcode Analysis Reversing & Emulation Framework. With SHAREM it is completely unnecessary to ever execute or debug a shellcode, yet more than 20,000 WinAPIs and virtually all user-mode Windows syscalls can be enumerated, identifying function calls, with both parameters and return values identified. SHAREM is the first tool to provide support for Windows syscalls. Second, SHAREM provides extremely accurate disassembly. This is firstly accomplished through static analysis, but it may be enhanced to a nearly flawless state by integrating emulation data to the already highly accurate disassembly. Given that one of SHAREM's unique features and greatest innovations is complete code coverage, this can help achieve greater accuracy with shellcode. Third, SHAREM can not only successfully deobfuscate shellcode, but it can also merge the decoded form into the disassembly. It accomplishes this through a novel algorithm that captures the shellcode's true form, even if the shellcode would later then re-obfuscate itself.

The rest of the paper is designed as follows: In section 2, we introduce related work. In section 3, we present the emulation architecture for SHAREM, and in section 4, we introduce its disassembly architecture. In section 5, we provide validation, and in section 6 we discuss contributions before concluding in the final section.

2 Related Work

Currently, if one wishes to perform low-level analysis of shellcode, the most useful method is a hybrid approach of both manual static and dynamic analysis. Tools such as dissemblers, like IDA Pro, Ghidra, Binary Ninja, and Radare2, may be used to provide disassembly. Debuggers such as WinDbg, x64dbg, OllyDbg, and Immunity can be used for dynamic analysis, showing the shellcode in execution. However, to be executed, the shellcode must be part of a PE file, requiring it to be placed in a harness file, which must be compiled. Two industry tools, Scdbg and Speakeasy [2, 3], provide limited emulation of shellcode, giving a high-level listing of APIs; previous research [2, 3] also provided limited emulation or facilitated dynamic analysis. However, the latter two have never been publicly available; both are more than a decade old.

Spector [4] was a shellcode emulator that used symbolic execution to obtain API and parameters from the shellcode, providing a high-level overview of functionality. Spector could create a limited listing of API calls and parameters. While Spector was not released publicly and unavailable to examine, it has influenced subsequent research.

While not a shellcode emulator, Shellzer [5] was able to dynamically analyze shellcode, and like Spector it has been publicly unavailable. It was capable of creating a listing of the numerated API functions and parameters; additionally, it created a listing of retrieved URLs. Shellzer relied upon the PyDbg Python debugging library for Windows to single-step through code, rather than executing the shellcode directly. To determine if a function was being called, it would check to see if the program counter pointed to memory from a DLL; if it did, it could then check to see if that corresponded to an API. Shellzer would attempt to simulate success by modifying certain parameters and return values. Although it was not an emulator, Shellzer was effective at determining API's utilized.

One Linux effort [6] at shellcode detection and emulation was to create snapshots of a process's virtual memory prior to it being given input; these snapshots were processed through a shellcode detector. An emulated environment of the process and registers was created from snapshots, allowing any found shellcode to be monitored by a malicious behavior detector. The analyzer also was able to emulate instructions and determine the functions called and arguments.

Scdbg [2] is an industry shellcode analyzer and emulator, identifying APIs and parameters used, with support for 246 opcodes, 15 DLLs, and 242 APIs. Scdbg provides a limited debugger, allowing a user to step through emulated execution of shellcode. While highly useful, this older tool is constrained by its partial support for APIs, DLLs, and opcodes. Additionally, we have found failure points with more complex shellcode. For our own efforts, we had used more hands-on approaches to analyzing shellcode, and we had never used Scdbg, until we were many months into SHAREM.

Speakeasy [3] is a framework by FireEye for the emulation of Windows kernel and user mode malware. SpeakEasy was released after work had begun on SHAREM and after SHAREM had been funded via an NCAE/NSA research grant. As such, we only used it very minimally to test shellcode samples, to compare its efficacy to that of SHAREM's. While not devoted to shellcode, this recent framework allows for some emulation of shellcode, identifying APIs, their parameters, and results. Speakeasy has a memory manger, allowing for memory allocations by the shellcode to be saved. An emulator for user mode or kernel malware, Speakeasy achieves the same as previous [2, 4, 5] shellcode analyzers, creating a listing of APIs, parameters, and network indicators. In our testing, some complex shellcodes cause emulation to cease after just a couple API calls due to memory errors. As with all other tools, this tool cannot identify Windows syscalls.

https://t.me/learningnets

Detection of shellcode is intended to identify but not analyze shellcode. Early detection efforts for shellcode would simply look for NOP sleds [7]. This behavior could be avoided easily by attackers, thus avoiding detection; efforts would grow in sophistication. Two general approaches to shellcode detection are static analysis and dynamic analysis. With static analysis, detection efforts could be made by looking for specific patterns [8, 9]. While static analysis can be partly effective, it also is limited. Static analysis is not always reliable, due to obfuscation, which can result in inaccurate disassembly [6]. Sometimes it may not be possible to detect program behavior from shellcode without engaging in labor-intensive dynamic analysis. Yara signatures could be used to detect specific byte patterns of shellcode, merely to indicate shellcode is present, without being able to indicate shellcode functionality. Dynamic analysis can involve allowing the shellcode to execute in a sandbox environment if placed in a harness file or through emulation of network activity [10, 11].

Network detection of shellcode [12–15] can be effective, although detection can be limited in that it is mostly based on rules, as with Suricata and Snort, though CPU emulation [10] is possible. Signatures may not always be useful, due to polymorphic behavior, allowing shellcode to transform dramatically [12, 16], as with encoded shellcode that decodes itself. SigFree [17] proposed data flow analysis that entailed searching for patterns that used *push* and *call* instructions to detect shellcode, without any use of signatures.

3 Architecture of SHAREM's Emulator

SHAREM's architecture is comprised of several elements. First, the shellcode instruction detection engine can statically discover numerous categories of instructions associated with shellcode or malware. SHAREM's emulator is perhaps the most important component, as it can hook more than 20,000 WinAPI calls, in addition to virtually all Windows syscalls. Not only can it discover APIs and syscalls, but it can also discover associated parameters, as well as the contents of different structures associated with said parameters. SHAREM has multiple novel contributions with respect to shellcode emulation. It can take emulation data and integrate that into the disassembler, providing for virtually flawless disassembly. SHAREM has a novel algorithm to capture the decoded form of a shellcode, sending it to the disassembler. SHAREM's emulator also has functionality for timeless debugging, recording not only every CPU instruction executed, but also the CPU state before and after each instruction. SHAREM's disassembly analysis engine can statically analyze the shellcode, generating a disassembly that is significantly more accurate than that which is produced by leading disassemblers, such as IDA Pro, Ghidra, Binary Ninja, etc. Additionally, the disassembler can integrate emulation data to enrich and enhance the disassembly found statically. SHAREM also contains several other components, which are outside the scope of this paper, but will be briefly mentioned here. The brute-force deobfuscator can decode shellcode, both with or without a decoder stub, and without utilizing emulation or any dynamic techniques, and it has support for parallelization and distributed computing. It is limited to combinations of simple encoding operations, such as *add, sub, xor, not, rol,* and *ror*, with each encoding operation utilizing one key. While immensely powerful, often emulation can allow for deobfuscation to occur naturally, and to do so far more expediently than could be achieved through brute force deobfuscation.

3.1 SHAREM Emulator

Emulation lies at the heart of SHAREM, serving myriad purposes. Firstly, SHAREM can discover WinAPIs and Windows syscalls. SHAREM is also able to uniquely integrate emulation data, thereby providing superior disassembly.

Unicorn-Engine. SHAREM utilizes Unicorn-Engine for emulation support. While Unicorn-engine can provide a virtual memory interface, it is necessary for very significant work to be performed to build appropriate Windows structures, to closely mirror what is done in both 32- and 64-bit environment.

Setup of SHAREM. SHAREM operates in a way that is unlike any other emulation tool that we are familiar with. SHAREM will actually harvest and utilize 31 Windows DLLs, which are placed into emulated virtual memory. The original DLLs are not modified, and indeed it would not be possible for us to modify the DLLs in the system directories. They are instead copied to SHAREM directories. SHAREM has an algorithm to inflate the DLLs, to mimic what occurs naturally. Without inflating the DLLs, much emulation would fail. While it would not be difficult to instead provide a listing of necessary values for different DLLs and other internal Windows features, that would be ineffective for all shellcode. Some shellcode can behave in a highly unorthodox fashion, and attempts at simulating values found in the PE file may not always work. Thus, to ensure a much higher degree of accuracy, we provide the actual PE files. SHAREM will harvest these PE

https://t.me/learningnets

files only once, both for 32- and 64 bit architectures, when emulation is first attempted. This process may take several minutes to complete, but it need not be done again. SHAREM will also utilize the Pefile Python library, to parse each DLL. This identifies both functions present as well as corresponding addresses. These can be calculated to match with our mappings of the DLLs and memory, and thus functions and virtual memory addresses are saved into a Python dictionary. For Linux, it is not possible to harvest DLLs from the OS. Those must be supplied directly. Currently we do not provide harvested DLLs that have been inflated, although we may do so at some point in the future.

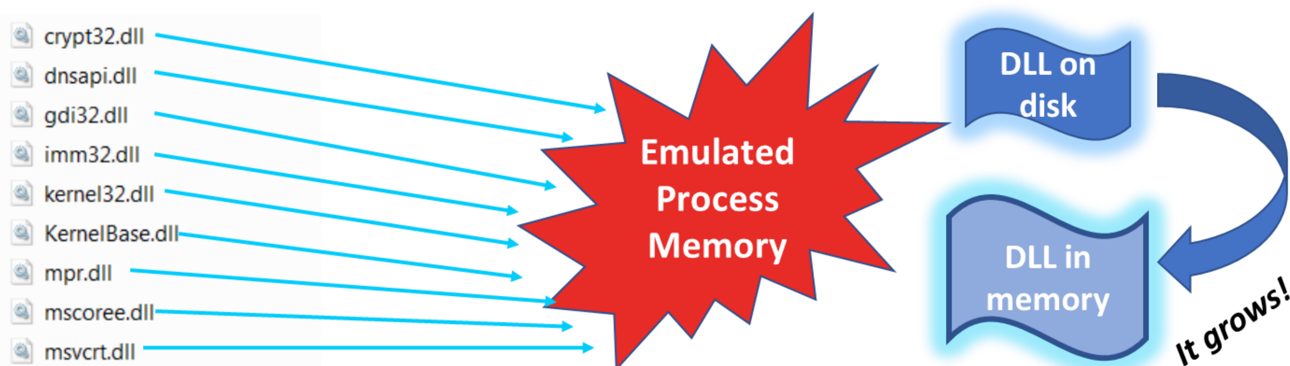

Fig. 1. SHAREM extracts and inflates DLLs from disk, emulating what is done by Windows, placing them into process memory.

3.2 Emulation of WinAPI Functions

SHAREM can emulate WinAPI's. For emulation to be possible, several Windows internal structures must be created, including the TEB, the PEB, the PEB LDR, and all the module doubly linked lists. The flinks and the blinks are used to point to the next and previous entries in a doubly linked list; each must be initialized with appropriate values, so that all DLLs are linked together. At the present time of writing, SHAREM has 31 DLLs, which are always in process memory, and each is linked with other DLLs in the doubly linked lists. These 31 DLLs allow for approximately 20,000 WinAPIs[1] to be supported. It is very likely that more will be supported in the future. because all these DLLs are and process memory, then it is likely that shellcode emulation will not fail, even if the shellcode attempts something unorthodox, while traversing the structure of the PE file format.

It is common knowledge among reverse engineers and malware analysts that very often the PEB is abused to find addresses of DLLs. Once the address of the DLL is found, then properties of the PE file format structure can be applied, allowing for the exports directory to be reached, and runtime addresses to be obtained for functions. There can be naive as well as sophisticated variations on this approach, but ultimately one of the goals of shellcode initially is to discover runtime addresses. These can then be stored in memory, able to be easily retrieved when the shellcode author wishes to call a function with its appropriate parameters.

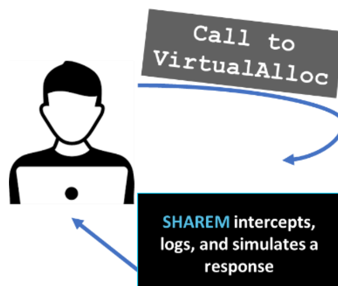

Fig. 2. SHAREM can intercept all function calls, logging them and simulating an appropriate response.

Lookup Dictionaries for Functions. SHAREM has sets of original lookup dictionaries for all DLLs that are supported. Each provides complete function prototype information. With this we can easily support thousands of APIs, allowing for

[1] A previous total of 16,000 was at one point shared, but that was owing to miscalculation.

https://t.me/learningnets

SHAREM to correctly emulate the shellcode. This can indicate the number of parameters to remove from the stack or x64 registers, in order to log the data correctly according to the number of parameters. For instance, SHAREM can in an automated fashion detect and extract strings, as that is indicated by parameter type. Thus, custom logic to perform this task is not required for each and every function parameter that may contain a string. By and large, this automatic approach has an extremely high level of accuracy.

In order to utilize the lookup dictionaries, SHAREM will capture the runtime address for the function the shellcode is attempting to call. SHAREM will look that address up in a dictionary, to match it with the name of the function. That function name can then serve as a key to lookup function parameters, as can be seen in the figure below.

```
dict4_advapi32 =  {'GetUserNameA': (2, ['LPSTR', 'LPDWORD'], ['name', 'size'], 'BOOL'), 'GetUs
    'size'], 'BOOL'), 'GetCurrentHwProfileA': (1, ['LPHW_PROFILE_INFOA'], ['pInfo'], 'BOOL'),
    ['LPHW_PROFILE_INFOW'], ['pInfo'], 'BOOL'), 'IsTextUnicode': (3, ['LPCVOID', 'INT', 'LPINT
    'AbortSystemShutdownA': (1, ['LPSTR'], ['lpMachineName'], 'BOOL'), 'AbortSystemShutdownW':
    'InitiateSystemShutdownExA': (6, ['LPSTR', 'LPSTR', 'DWORD', 'BOOL', 'BOOL', 'DWORD'], ['I
    'bForceAppsClosed', 'bRebootAfterShutdown', 'dwReason'], 'BOOL'), 'InitiateSystemShutdownE
```

Fig. 3. SHAREM provides lookup function dictionaries, to help automate discovery of thousands of functions that may not require special handling.

Custom Implementations for Functions. Many functions are relatively simple or may have no security ramifications, and the lookup dictionaries are sufficient to handle the API. In other cases, the WinAPI function may be highly complex, and it may be desirable to simulate aspects of what it is doing. For instance, with some registry functions, we may wish to create an entry for a registry key in our Registry Manager, which attempts to simulate the registry. Or, perhaps the function requires an allocation of memory to be created. Or, with UrlDownloadToFileA, the API may attempt to download a file. If it succeeds, the file is hashed, and it is saved to a pseudo filesystem. If there is nothing to be downloaded, then it will simulate success, using predetermined values, which the user can customize in the config file or in the UI. So, while it is possible for some APIs to be emulated with just the lookup dictionaries, we also have hundreds of custom implementations to allow for more elaborate setups.

```
0x120000b4 WinExec(LPCSTR lpCmdLine, UINT uCmdShow)
        LPCSTR lpCmdLine: bitsadmin.exe /transfer /Download /priority Foreground http://127.0.0.1:80/note.txt
C:\thefile.txt
        UINT uCmdShow: SW_HIDE
        Return: INT 0x20

0x120000c3 ExitProcess(UINT uExitCode)
        UINT uExitCode: ERROR_SUCCESS
        Return: void None
```

Fig. 4. SHAREM can emulate WinAPIs, logging functionality that it intercepts. Human readable equivalents are logged in place of hexadecimal values.

Human Readable Equivalents. Whenever possible, SHAREM will log the human readable equivalent to a hexadecimal value. For instance, 0x40 may be well known to many, indicating PAGE_EXECUTE_READWRITE. Thus, that is what would be logged. However, SHAREM has several hundred human readable equivalents, many of which may not be known even to the most seasoned reverse engineers. These must be provided as part of the custom implementations of functions. While SHAREM logged the human readable equivalents, it goes without saying that the original hexadecimal value is what is utilized within the emulation.

Matching Handles to their Human Readable Equivalents. SHAREM also maintains a handle manager to map out handles to their human readable equivalents. Thus, while the emulation will internally utilize hexadecimal handles, when it is logged, it maps it out to the human readable equivalent. An example could be a hexadecimal handle to a registry key. SHAREM would display the actual key that the handle corresponded to. This could be immensely useful, saving the user from what could possibly be tedious tracing, depending on the shellcode's level of activity.

https://t.me/learningnets

Simulating Success for Functions. As SHAREM is not a debugger, we are not actually executing the code contained within functions. SHAREM simply intercepts the emulated function call, logging parameters and simulating success**.** Shellcode may attempt to discern whether or not a previous function call was successful, such as looking for the S_OK (0x00) value, returned in EAX or checking certain parameter values. If the function call was not successful, then the shellcode would prematurely terminate. Thus, since the goal was to discover as much functionality as possible, SHAREM always simulates success. When implementing each function, thought is given is to how to best simulate success. For functions that utilize only lookup dictionaries, the default behavior is to return S_OK in EAX. There are also separate dictionaries that contain function success values. Custom implementations of functions may provide much more elaborate simulations of function success, as dictated by the requirements of particular functions.

Checksums. Experienced malware analysts are well acquainted with the fact that often shellcode may not directly utilize unencoded Assembly, while loading a function's runtime address. Instead, it may traverse through the exports directory, generating checksums for all function names. The shellcode may have a precomputed checksums for a set of specific functions, and the shellcode may compare the checksums until it finds a match. Once a match is found, then that index can be used in a corresponding array to obtain the function's runtime address. Thus, due to the reliance on checksums, in many cases emulating or debugging a shellcode may be the most reliable way to determine which function is being called.

Capturing Structures and Unions. Whenever possible, SHAREM will capture structures used as parameters for functions. In order to capture structured data, the particular structure must be added to the custom implementation for the function; it is not an automatic process. Any conceivable structure could be captured, including even structures that have other structures as members or that may contain unions. With unions, more than one parameter may share the same memory. A function parameter for a structure is a pointer to the actual structure. SHAREM can then go to that location and extract all the data, based on what is defined in the custom implementation. SHAREM displays structures and the output with indentations.

```
************* Syscalls *************

0x1200034b NtCreateKey(PHANDLE KeyHandle, ACCESS_MASK DesiredAccess, POBJECT_ATTRIBUTES ObjectAttributes, ULON
G TitleIndex, PUNICODE_STRING Class, ULONG CreateOptions, PUNLONG Disposition)
        PHANDLE KeyHandle: 0x16fffae8 -> HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run
        ACCESS_MASK DesiredAccess: 0xf013f
        POBJECT_ATTRIBUTES ObjectAttributes:
                ULONG Length: 0x18
                HANDLE RootDirectory: 0x0
                PUNICODE_STRING ObjectName: \Registry\Machine\Software\Microsoft\Windows\CurrentVersion\Run
                ULONG Attributes: OBJ_CASE_INSENSITIVE
                PVOID SecurityDescriptor: 0x0 -> 0x0
                PVOID SecurityQualityOfService: 0x0 -> 0x0
        ULONG TitleIndex: 0x0
        PUNICODE_STRING Class: 0x0
        ULONG CreateOptions: 0x0
        PUNLONG Disposition: 0x0
        Return: NTSTATUS STATUS_SUCCESS
        EAX:  0x1d - (Windows 10, SP 21H1)
```

Fig. 5. In this Windows syscall, we can see a complete structure as a parameter, indicating that persistence is being set.

3.3 Emulation of Windows Syscalls.

Prior to the advent of ShellWasp [1], it was extremely rare for Windows syscalls to be used in shellcode for purposes other than egghunters [18, 19]. An egghunter is a specialized shellcode designed to find a larger shellcode or egg injected into process memory at an unknown location. To find the egg, a syscall such as NtAccessCheckAndAuditAlarm is used to check if memory is valid, before checking that memory location for a distinctive tag, e.g. w00tw00t. Once the tag is found, control flow can transfer to the newly found egg. While egghunters have been widely used for nearly two decades, the portability problem associated with Windows syscalls prevented their widespread use. As part of this research for SHAREM, we undertook an exhaustive study, trying to discover historical uses of Windows syscalls in shellcode. We were only able to find one relic from 2005 [20], where syscall system service numbers (SSNs) were hardcoded in a Windows

https://t.me/learningnets

XP-era shellcode. Thus, one important research question was how can we emulate Windows syscalls, when there are really no current samples whatsoever. Was SHAREM to be a rare case of defenders being ahead of adversaries? To properly emulate Windows syscalls, extensive reverse engineering and research was performed. To test SHAREM for efficacy, it was necessary to create samples that utilized Windows syscalls. That led to a presentation at DEFCON 30 and the creation of ShellWasp at Black Hat Middle East and Africa 2022. ShellWasp provided an efficient method to overcome the portability problem associated with using syscalls, and special parts of the PEB must be initialized, in order for the ShellWasp technique to work.

```
************* Syscalls *************

0x12000008 NtAllocateVirtualMemory(HANDLE ProcessHandle, PVOID *BaseAddress, ULONG_PTR ZeroBits, PSIZE_T RegionSize, ULONG AllocationType, ULONG Protect)
        HANDLE ProcessHandle: 0xffffffff
        PVOID *BaseAddress: 0x16fffd80 -> 0x25000000
        ULONG_PTR ZeroBits: 0x0
        PSIZE_T RegionSize: 0x16ffffe8 -> 0x1000
        ULONG AllocationType: 0x30000
        ULONG Protect: PAGE_EXECUTE_READWRITE
        Return: NTSTATUS STATUS_SUCCESS
        EAX:  0x18 - (Windows 10, SP 21H1)
```

Fig. 6. SHAREM can emulate complex syscalls, providing detailed information for users.

SHAREM initializes the address pointed to by fs:0xc0 to point to a special location that is always hooked for syscalls. With syscalls, the SSN is stored in the eax register, and this is extracted and matched to determine the Windows syscall being used. The SSNs for Windows syscalls can change with each new OSBuild, and there are more than thirteen OSBuilds for Windows 10 alone. Thus, SHAREM allows the user to enter the desired Windows OSBuild, via the UI or the config. Again, this is necessary as SSNs may change across multiple builds. A particular build will determine the number of parameters to be taken from the stack. Thus, thus if the user wishes to test for a different OSBuild, he or she must re-run the emulation. Practically speaking, most users will utilize automatic Windows updates, and it seems likely most new shellcode samples with syscalls likely would target the current or a very recent OSBuild, rather than one that is years out of date. In addition, while SSNs change with each new OSBuild, many times they stay the same across multiple builds. Thus, there may be little to no difference between the two most recent OSBuilds.

In testing Windows syscalls, we created a number of these painstakingly, both for Windows 7 and Windows 10, reverse engineering them from normal usage in Windows applications. In general, Windows syscalls require significantly more complex setup than WinAPI functions. As malware authors grow in sophistication, being able to detect and fully log all parameters for Windows syscalls represents an important, cutting-edge contribution.

Sources for SSNS for Different OSBuilds. Whenever possible we use existing syscall tables [21]. However, we create our own syscall tables for the OSBuilds, covering approximately the last two years, and these include the most recent OSBuilds for Windows 10 and 11. It is our intention to support future OSBuilds.

Pseudo-Emulation of Windows Syscalls. In a very early period of development for SHAREM, we created a special algorithm to determine all possible variations of calling a Windows syscall in Assembly. Because syscalls are moved into EAX, and this is often abundantly clear, we can definitively determine what an SSN is, for a given instance of calling a Windows syscall. In some cases, the value in EAX is not immediately clear with the pseudo-emulation, and thus SHAREM will consider it "unknown." With this pseudo-emulation, the user can determine all the possible Windows syscalls being called for a particular instance. However, it is not possible with the pseudo-emulation to determine parameter values. This pseudo-emulation is effective both with shellcode and PE files. In some cases, the pseudo-emulation may produce false positives. This feature has in many ways been superseded by the true emulation of Windows syscalls, although it retains some unique capabilities that cannot be achieved with true emulation, such as identifying all possible SSNs for a single instance or working with PE files.

Lookup Dictionaries for Windows Syscalls. Similar to Windows functions, there are lookup dictionaries for Windows syscalls. Unlike APIs, syscalls are not well documented by Microsoft, and in fact only about 30 syscalls are documented

https://t.me/learningnets

on their website. However, there are many alternative sources of function prototypes for Windows syscalls. Some belong to sites like NTAPI Undocumented Functions, but between those two sources, more than half of all user-mode Windows syscalls were not documented in one site. Thus, as part of this research, each syscall was searched for individually, until a functional prototype could be found for it. Some of these are the results of various reverse engineering efforts, so the parameter names and types may be good faith approximations. All of this data was collected and aggregated, creating a dictionary of Windows syscalls, each containing necessary function prototype data, similar to lookup dictionaries for APIs. Later we learned of a source that had already compiled much of this data, so some of our efforts could have been avoided.

Logging Artifacts. SHAREM automatically extracts numerous categories and subcategories of data. These are extracted from API parameters using regular expressions. This also means that if a parameter uses a handle, and SHAREM maintains a human readable equivalent to that handle, such as a registry key, then the key is what would be subjected to the regular expressions. By default artifacts are logged immediately after emulating a shellcode. They are also printed in the text and JSON reports that can be generated.

```
************* Artifacts *************
*** Paths ***
** Misc **
c:\temp
c:\temp\ChromeUpdates.bat
c:\temp\ChromeUpdates.bat

*** Files ***
** Misc **
cmd.exe
urlmon.dll
default.css
ChromeUpdates.bat

*** Command Line ***
cmd.exe /c sc stop WinDefend
cmd.exe /c netsh advfirewall set allprofiles state off
cmd.exe /c md c:\temp\
cmd.exe /c net user TestUser Open24X7! /add && net localgroup administrators TestUser /add
cmd.exe /c sc create ChromeUpdater binpath= c:\temp\ChromeUpdates.bat start= auto && sc start ChromeUpdater

*** Web ***
http://167.99.229.113/default.css
```

Fig. 7. SHAREM can log many categories and subcategories of useful artifacts from shellcode.

Dealing with Infinite Loops. Infinite loops can be disastrous for emulation, if a shellcode is caught in a loop**.** While breaking out of loops is not a novel concept, it is one that is fairly important and that is implemented by SHAREM. By being able to break out of loops, we can ensure that additional functionality is discovered. SHAREM Provides an upper limit that can be used to specify when loop iterations can be broken out of. SHAREM tracks all jump conditionals, and if a jump conditional is repeated up to a particular threshold, SHAREM can invert the logic. It changes EIP to another branching location, while resetting the threshold count.

Complete Code Coverage. While this topic will be explored more fully elsewhere, SHAREM provides a novel contribution with complete code coverage. This optional feature can allow all paths to be explored and emulated, even those which otherwise would be unreachable.

3.4 Timeless Debugging.

Timeless debugging [22] was not intended to be a primary feature of SHAREM, but it is one that can be of great practical value. SHAREM was able to capture all instructions executed, both before and after each executed assembly instruction. SHAREM takes a snapshot of the current CPU state, showing the contents of registers. SHAREM outputs all data to a text file, which may be reviewed subsequently. Timeless debugging can be immensely beneficial, in understanding the often

https://t.me/learningnets

convoluted trajectory of a shellcode, particularly if it is behaving in a way we may not anticipate. As shellcode samples often have loops that may repeat thousands of times, timeless debugging can provide an efficient alternative to having to debug in the traditional fashion with breakpoints.

```
23      >>> EAX: 0x0 EBX: 0x4b1ffe8e ECX: 0x0 EDX: 0x12000000 ESI: 0x0 EDI: 0x0 EBP: 0x17000000 ESP: 0x16fffff4
24  0x1200011e  xor edi, edi
25
26      >>> EAX: 0x0 EBX: 0x4b1ffe8e ECX: 0x0 EDX: 0x12000000 ESI: 0x0 EDI: 0x0 EBP: 0x17000000 ESP: 0x16fffff4
27  0x12000120  mov edi, dword ptr fs:[0x30]
28
29      >>> EAX: 0x0 EBX: 0x4b1ffe8e ECX: 0x0 EDX: 0x12000000 ESI: 0x0 EDI: 0x11017000 EBP: 0x17000000 ESP: 0x16fffff4
30  0x12000127  mov edi, dword ptr [edi + 0xc]
31
32      >>> EAX: 0x0 EBX: 0x4b1ffe8e ECX: 0x0 EDX: 0x12000000 ESI: 0x0 EDI: 0x11020000 EBP: 0x17000000 ESP: 0x16fffff4
33  0x1200012a  mov edi, dword ptr [edi + 0x14]
```

Fig. 8. Timeless debugging can give many insights as to the behavior of a shellcode, allowing for a more refined understanding of its behavior.

3.5 Detecting Instructions through Emulation

While SHAREM has its own disassembler that is purely static analysis, SHAREM also was able to capture emulation data that can later be integrated with the disassembler. The default entry point for shellcode is offset 0, although the user may change that to anything they wish. Assuming the entry point is correct, then if we can emulate instructions, we can conclusively know that those bytes are instructions. More importantly, we know the beginning offset of each instruction and the size of the instruction. This metadata is saved and can be used to override what may otherwise have been determined through static analysis. Thus, assuming the emulation is accurate, then the quality of the disassembly will have been greatly enhanced, as we can be guaranteed that every emulated instruction will be correct, assuming there were no special issues with the emulation, which would be highly unusual. SHAREM does not save data on specific assembly instructions that are executed. It only captures metadata that can allow for correct disassembly to later be generated by overriding what would otherwise be produced by the SHAREM disassembly analysis engine (DAE). In more practical terms, by integrating emulation data, we can achieve sometimes nearly flawless disassembly.

3.6 Detecting Data through Emulation

SHAREM can accurately distinguish between data and instructions through emulation in many cases. Data often will be read to or written to, although with self-modifying code, then a majority of all bytes will be written to and read. Fortunately, SHAREM can detect self-modifying code, and it handles those differently. Thus, if there are bytes that are used exclusively as something that is read to or written to, then they are classified as data. SHAREM can also perform analysis to detect certain types of data, such as API pointers and in API table. SHAREM will not only classify those as data, but it will also indicate the API that it eventually will be pointing to. Indeed, API tables are common in shellcode. SHAREM has its own custom string detection algorithms, and strings that are found are also classified as data. In some cases, strings may be erroneously classified as data, in spite of emulation, but likely future updates will address this. With self-modifying code, because the majority of bytes are being both written to and read, as well as being executed as instructions, SHAREM simply maintains a listing of the number of times each instruction is read or written to. It may require that all bytes be read or written to at least twice to be considered data or in some cases even thrice to be considered data. This approach allows for data to still be detected with self-modifying code.

https://t.me/learningnets

```
0x1e4 call eax                          ff d0                   ..
    ; call to VirtualAlloc
     (0x0, 0x1000, MEM_COMMIT,
      PAGE_EXECUTE_READWRITE)
0x1e6 ret                               c3                      .

0x1e7 dd c78a3146                       c7 8a 31 46             ..1F
0x1eb dd 192b9095                       19 2b 90 95             .+..
0x1ef dd 2680acc8                       26 80 ac c8             &...
0x1f3 dd f572993d                       f5 72 99 3d             .r.=
0x1f7 dd fe6a7a69                       fe 6a 7a 69             .jzi
0x1fb dd ffff0000                       ff ff 00 00             ....
0x1ff CreateProcessA - API pointer         00 00 00 00             ....
0x203 ExitProcess - API pointer    01 00 00 00             ....
0x207 LoadLibraryA - API pointer           02 00 00 00             ....
0x20b Sleep - API pointer          03 00 00 00             ....
0x20f VirtualAlloc - API pointer           04 00 00 00             ....
0x213 dd 99235dd9                       99 23 5d d9             .#].
0x217 dd ffff0000                       ff ff 00 00             ....
0x21b URLDownloadToFileA - API pointer     05 00 00 00             ....
0x21f urlmon.dll ; string                  75 72 6c 6d 6f 6e 2e 64 ... urlmon.dll.
```

Fig. 9. SHAREM allows for data to be accurately distinguished from instructions. In the above, only the first two lines are code, while the rest is data, which SHAREM displays in different ways.

3.7 Emulation of Encoded Shellcode

While SHAREM has brute-force deobfuscation capabilities, sometimes the easiest method is to allow the shellcode to execute and decode itself in memory. SHAREM is effective with both simple and sophisticated encoding schemes. A shellcode naturally decodes itself with the help of a decoder stub. This can allow for APIs and syscalls to be logged. SHAREM uses fuzzy hashing, SSDeep, to determine if a shellcode is encoded or self-modifying. If it is, then it treats it differently than if it were not. SHAREM also uniquely can capture the original form of a shellcode. It does with a novel algorithm. That is to say, SHAREM doesn't just simply take a snapshot of the shellcode after execution ends. After all, it is possible for a shellcode to reencode itself, to conceal what it is doing. Thus, SHAREM takes three primary snapshots. The first is the starting form of the initial encoded shellcode. The second is the executed form of the shellcode, and that is done only for each instruction that is executed. It captures the starting location, the size of the instruction, and the bytes that constitute the instruction. Thus, if a shellcode were to reencode the bytes that comprise that instruction, then it would not matter because the executed form had already been preserved. Accordingly, SHAREM may take hundreds of thousands of many snapshots during the execution of the shellcode. Finally, SHAREM takes a snapshot of the final form of the shellcode after emulation. All of the above are merged together using a special formula. The executed form of each bite is prioritized over all else, and then the final form is prioritized over the starting form.

The end result is often a shellcode whose decoded form can be revealed through disassembly. At no point does SHAREM simply capture assembly instructions that were executed and somehow aggregate them together. Instead, SHAREM captures the correct state of each byte with appropriate metadata, while recording whether each is and instruction or data, along with corresponding metadata, such as the size of instruction and starting offset of instruction or data.

https://t.me/learningnets

```
0xa6 call dword ptr [edx + 0x14d]            ff 92 4d 01 00 00      ..M...
    ; call to WinExec
      (mshta file://C:\Users\Public\evil.hta, SW_HIDE)
0xac pop edx                                 5a                     Z
0xad pop ebp                                 5d                     ]
0xae push ebp                                55                     U
0xaf push edx                                52                     R
0xb0 xor eax, eax                            31 c0                  1.
0xb2 push eax                                50                     P
0xb3 call dword ptr [edx + 0x145]            ff 92 45 01 00 00      ..E...
    ; call to ExitProcess
      (ERROR_SUCCESS)
         label_0xb9:
0xb9 cld                                     fc                     .
0xba xor edi, edi                            31 ff                  1.
0xbc mov edi, dword ptr fs:[0x30]            64 8b 3d 30 00 00 00   d.=0...
    ; load TIB
0xc3 mov edi, dword ptr [edi + 0xc]          8b 7f 0c               ...
    ; load PEB_LDR_DATA LoaderData
0xc6 mov edi, dword ptr [edi + 0x14]         8b 7f 14               ...
    ; LIST_ENTRY InMemoryOrderModuleList
```

Fig. 10. Although this shellcode was encoded, SHAREM displays the decoded form in the disassembler, both with functions and parameters identified.

4 Architecture of SHAREM's Disassembler

When starting work on SHAREM, there originally were no plans for a disassembler. However, as part of this research, we examined dozens of shellcode samples for which we possessed the source code in leading disassemblers, such as IDA Pro, Ghidra, and Binary Ninja. We were often horrified at how comically incorrect some of the disassembly was. This was chiefly due to the misclassification of data and instructions. Data would be classified and interpreted as instructions, while some other actual code would start at an incorrect offset, resulting in inaccurate disassembly. In an effort to deal with this, we decided to create my own disassembler – built only for shellcode. We recognize that shellcode often may not follow conventions, and that may play fast and loose with the rules. Indeed, data may be freely interspersed with instructions at often odd locations.

4.1 Disassembly Analysis Engine

Our approach then was one that was more empirical and based on experimentation. We would attempt to determine why something was misclassified as data or instructions. Some led to obvious solutions, such as searching for strings that may be inserted as data in a shellcode. Others were far more subtle, such as searching for hidden jumps or calls, or checking to see if the destination of different jumps or calls were possible. After all, if a shellcode attempted to go to a location that did not exist, then those instructions were not accurate, and more likely than not those bytes were actually data. All of the above and more could be accomplished through static analysis. The approach taken with this research is an iterative design cycle, using many modern Windows shellcode samples as guides, allowing for refinement of approach for disassembly analysis functions. Once inaccurate disassembly was determined, efforts were then made to discover the root cause. By addressing the root cause, then other future misclassifications could be avoided for that entire category or subcategory. This approach could be described as coming up with a set of rules for handling of disassembly. This led to many special analysis functions being developed that could help implement some of these rules. While this approach could never be perfect, it often resulted in significantly more accurate disassembly. With samples for which we possessed the source code, the disassembly could be as much as 40% inaccurate, whereas with SHAREM, the disassembly could be as much as 95% accurate.

The approach taken with this research is an iterative design cycle, using many modern Windows shellcode samples as guides. Once inaccurate disassembly from DAE was found, the root cause was determined. Analysis functions were then created or refined, until the inaccuracy could be handled. This led to the creation of analysis phases to correctly distinguish between data and instructions in the numerous Windows shellcode examples we analyzed.

3.2.1. DAE Classification of Bytes

https://t.me/learningnets

SHAREM maintains complex metadata regarding all bytes in a shellcode sample. The most important metadata is whether or not a particular byte is data or code. If a byte is data, we might indicate the type of data, such as strings, API pointers, etc., and then the data would be represented in a special way that corresponds to the data type. No metadata is stored on what the Assembly instructions should be. Instead, SHAREM stores metadata that can be used to generate chunks of disassembly. For instance, there might be 29 bytes of code followed by some data. If those 29 bytes were uninterrupted instructions, then we could simply begin disassembling at the start of those 29 bytes, and the resulting disassembly should be accurate. A particular shellcode may go back and forth between data and code numerous times. If each byte is classified correctly as data or instructions, whether by the static analysis DAE or by integrating emulation data, then it should always produce more or less error-free disassembly.

It is extremely common with shellcode to have much data, such as cryptographic checksums or strings, be freely intermixed with code, often in a way that is not readily predictable. This compels us to take a more flexible and different approach to disassembly with Windows shellcode, than we might otherwise do.

The following are a series of analysis phases that can be completed by SHAREM's DAE, to try to gain more accurate disassembly. These analysis phases result in the creation or modification of different metadata, which later then is used to generate the disassembly. If emulation data is integrated into SHAREM, then it will always override and replace what the metadata produced by DAE might indicate.

Finding FF Bytes. Shellcode may sometimes have large chunks of repeating FF or 00 bytes. However, these bytes, particularly FF bytes, may form legitimate assembly instructions or be part of a negative number formed by two's complement. If it is a long series of repeating FF or 00 bytes, then we would classify those as data. SHAREM will analyze repeating FF bytes, checking it against known instructions that utilize FF. If those are found, we assume they are instructions, and those are excluded. for groups of four or more repeating FF or 00 bytes, we assume that those are data, and we classify them accordingly.

Checking for Valid Jump Destinations. One problem with many shellcode samples were erroneous jumps to destinations that were not possible to exist within the shellcode. Thus, one of two possibilities existed. First, those bytes should be part of other instructions. The more likely outcome is that they are data. Thus, for each set of bytes that are not considered data, SHAREM generates a chunk of disassembly to determine if its destination exists, which is effective with both positive and negative values. Accordingly, if the shellcode attempts to go to a location that is outside the realm of possibility, such as a *jmp 0x5000*, when the shellcode is only 192 bytes, then those bytes would be classified as data.

Finding Hidden Jumps and Calls. Shellcode may often have a series of extremely convoluted function calls and jumps in a tiny space. The end result is that when producing disassembly, particularly if there is no emulation involved, then some jumps or calls could be lost or misclassified. This is especially true if we have a lot of data freely intermixed with instructions. In examining shellcode samples, having hidden jumps or calls seemed to occur with some regularity. Accordingly, SHAREM will search for all short jumps or short calls, specifically those that begin with the opcodes EB, E8, or E9. If those short jump candidates are found, then each is evaluated to determine if the branching is in a negative or positive direction, and what specifically the destination address is. SHAREM then evaluates the destination address to see if it is plausible; if it is outside the realm of possibility, then we can safely regard that as a false positive. However, if the destination is indeed plausible, then those bytes will be marked as instructions, if they were not already. Accordingly, we would ensure an instruction began at that offset. In other words, it would not jump to the middle of another instruction. While this approach inherently can never be 100% accurate, in actual practice it does greatly improve the quality of disassembly, by finding short jumps or short calls that otherwise would be lost. And, arguably, branching instructions are some of the most important to find.

Handling Strings in Disassembly. Depending on the shellcode author, there may be plain text strings that are intermixed with code. Ordinarily most disassemblers would misclassify these as instructions. However, SHAREM has its own algorithms to detect strings, including some that may be formed in a slightly different fashion than what is often the case. Found strings can be classified as data and represented as strings, whether it is from ASCII or Unicode. While this may seem obvious, this has been an issue that has plagued the accuracy of many shellcode samples.

4.2 Integration of Emulation Data to Enhance Disassembly

SHAREM can produce disassembly independent of the utilization of emulation data. However, if the user decides to emulate the shellcode, then by default emulation data is obtained, and it will automatically be used. As described previously, SHAREM will obtain metadata on the starting location of an instruction, the size of the instruction, and the fact

https://t.me/learningnets

that it is code as opposed to data. Similarly, as described previously, SHAREM employs special techniques to determine if certain bytes are data. It is possible that some of SHAREM's original metadata is changed with emulation data.

The method by which SHAREM ensures that emulation data always overrides what is found through static analysis is simple but highly effective. Bytes may have already had their classification changed to data or code, but there still could be a possibility that disassembly of instructions could begin at an incorrect offset. Thus, SHAREM uses emulation data to ensure that each instruction begins and ends at the correct offset. Rather than perhaps generating a chunk of disassembly that is five lines long, before encountering data, SHAREM generates only one line of disassembly five different times before next encountering data. Additionally, if some bytes had been marked as data through static analysis, but the emulation data indicated they were actually code, then those bytes would be reclassified. They then would be treated in a similar fashion to the above. Again, it bears repeating that being able to accurately distinguish between data and instructions at every possible offset is most important with disassembly, and if we can achieve this then the disassembly should be free of errors.

4.3 Disassembly of Encoded Shellcode

One highly novel aspect of SHAREM is its ability to display the decoded form of an encoded shellcode. This process occurs automatically. As described previously, fuzzy hashing with SSDeep can determine if shellcode is self-modifying. If so, it uses a special algorithm to capture its decoded form. Again, this is not anything as simplistic as allowing a shellcode to decode itself in memory and then capturing its final form from memory. With SHAREM, different states of the shellcode are captured, allowing them to be merged, thereby preventing a shellcode from reencoding each instruction.

The ability to display disassembly of the deobfuscated form of an encoded shellcode is perhaps a game changer`. The implications of what could be done with that are truly far reaching. For instance, if someone found a shellcode online and wanted to utilize that in an exploit, they could check to see what the shellcode is actually doing and if there is anything extra that is hidden in there. Similarly, a shellcode found as part of malware could easily be reversed with this approach, without the need to single-step through shellcode in a debugger.

One feature of encoded shellcode is a decoder stub. The decoder stub can vary greatly in complexity, from simply modifying each byte with an XOR instruction and the same key, to sophisticated decoder stubs with many encoding operations, changing keys, and with junk instructions inserted for additional obfuscation. Generally, the decoder stub is not encoded, and in fact we are not aware of any decoder steps that are encoded, although it's possible one could have a decoder stub and then a second stage decoder stub. At any rate, we want to preserve the decoder stub, presenting that as part of the disassembly. SHAREM will automatically mark the end of the decoder stub in the disassembly, for additional clarity.

4.4 Disassembly Annotations

SHAREM is a complex framework with many features, not all of which are able to be documented here. As part of the initial analysis, SHAREM searches for what we call shellcode instructions. These are instructions associated with either shellcode or malware, such as GetPC gadgets, like *Call / Pop exx* or *Fstenv,* Heaven's gate instructions, *Push / Ret* gadgets, or the so-called PEB walking that inevitably occurs with Windows shellcode. The algorithms to find these capture unique data points, and some can find hundreds of possible variations, allowing for specific parts of the disassembly to be annotated. For instance, with PEB walking, we can't have distinct parts of Windows internal structures being identified with comments. Or, with a *Push / Ret* gadget, we might have both the *push* and *ret* identified. Additionally, SHAREM has its own algorithms to identify both ASCII and Unicode strings. If found, the string will appear in lieu of some other representation with a comment indicating that it is a string. With push stack strings, where a series of pushes can be used to create a string on the stack, those are also identified with comments. Finally, API pointers that can be found by analyzing memory after emulation will be indicated with a comment. Thus, SHAREM provides an extraordinary wealth of rich annotations, which appear automatically, surpassing what is done by other disassemblers.

```
0x120 mov edi, dword ptr fs:[0x30]        64 8b 3d 30 00 00 00    d.=0...
    ; load TIB
0x127 mov edi, dword ptr [edi + 0xc]      8b 7f 0c                ...
    ; load PEB_LDR_DATA LoaderData
0x12a mov edi, dword ptr [edi + 0x14]     8b 7f 14                ...
    ; LIST_ENTRY InMemoryOrderModuleList
```

Fig. 11. SHAREM can annotate key parts of the disassembly automatically, such as PEB walking features.

https://t.me/learningnets

```
0xaf push 0x74636500                    68 00 65 63 74      h.ect
0xb4 push 0x6a                          6a 6a               jj
0xb6 push 0x62                          6a 62               jb
0xb8 push 0x4f                          6a 4f               jO
0xba push 0x65                          6a 65               je
0xbc push 0x6c676e69                    68 69 6e 67 6c      hingl
0xc1 push 0x53                          6a 53               jS
0xc3 push 0x72                          6a 72               jr
0xc5 push 0x6f                          6a 6f               jo
0xc7 push 0x46                          6a 46               jF
0xc9 push 0x74696157                    68 57 61 69 74      hWait
          ; WaitForSingleObject - Stack string
```

Fig. 12. SHAREM's custom detection of Push Stack Strings is shown above. Here a string for an API is identified in the disassembler.

Beauty of Disassembly. SHAREM provides customizable options for display of disassembly. By default, opcodes are shown alongside any ASCII that can be generated from the opcodes. DAE vividly displays the disassembly with many colors and uniform spacing for optimal readability. The disassembly can also be exported as a text file and as a JSON file, allowing it to be easily imported into a web service that may use SHAREM. In the JSON, each element of the disassembly is separate, allowing for users to customize how they display the disassembly with different and unique colors, spacing, etc.

5 Ghidra Script to Ingest SHAREM's Output

While SHAREM provides excellent results with regards to shellcode analysis and emulation, one might want to adapt or automate further steps. Ghidra, the open-source software reverse-engineering framework created by the NSA, allows one to analyse binaries of numerous types and architectures. Shellcode for x86 and x86_64 is among the supported architectures, although the lack of context when statically dealing with a file requires Ghidra to make assumptions. SHAREM provides this context, based on the data it gathers during the emulation. The script for Ghidra to ingest SHAREM's emulation data resolves exactly this problem. This part of the paper will dive into the technical details of the Ghidra plugin script, what it does, and how it is best executed and used by a user.

5.1 SHAREM's Headless Execution

Since SHAREM can be executed headless (in an automatic manner where the command-line interface arguments suffice for its execution), one can execute the tool and wait for the emulation to finish. After this, the "jsondefaultdisasm.json" file can be used for any desired purpose. To figure below shows the first fourteen lines of the JSON file.

https://t.me/learningnets

```
1  {
2      "disassembly": [
3          {
4              "address": "0x0",
5              "instruction": "call 5",
6              "hex": "e8 00 00 00 00",
7              "size": "5",
8              "bytes": "CODE",
9              "dataType": "CODE",
10             "dataAccessed": "None",
11             "string": ".....",
12             "comment": "; ***Shellcode Entry Point, offset 0x0***",
13             "label": ""
14         },
```

Fig. 1. The SHAREM JSON holds many fields relevant to disassembly.

The “disassembly” field represents an array of objects, all of which have the same structure as the object which is outlined from line three to fourteen. The address is the offset from the start of the analysed shellcode file, in hexadecimal format. The instruction is the human readable form of the data (which is either an instruction or a piece of data), and the “hex” field is the hexadecimal representation of it. The size is the total size of the that particular instruction in bytes. The bytes field is either “CODE” or “DATA,” depending on what type it contains. The data type is a further specification, with cases such as “API Pointer” or “String.” Data accessed defines how the data is accessed, while the string field shows the string representation of the data. The comment field is, in combination with the address, the most important field in most cases, as it contains the comment SHAREM generated. This can be the value to an API call, as can be seen in the figure below.

```
351         {
352             "address": "0x4f",
353             "instruction": "call dword ptr [edx + 0x125]",
354             "hex": "ff 92 25 01 00 00",
355             "size": "6",
356             "bytes": "CODE",
357             "dataType": "CODE",
358             "dataAccessed": "None",
359             "string": "..%...",
360             "comment": "; call to GetProcAddress\n      (urlmon.dll, URLDownloadToFileA)",
361             "label": ""
362         },
```

Fig. 2. The comment field holds important textual information for different bytes.

The call to “[edx + 0x125]” is actually a call to GetProcAddress, which takes “urlmon.dll” and “URLDownloadToFileA” as arguments. As such, not only is the call to a register with an offset now clearly readable, the arguments for said call are also known during the analysis.

Ghidra Script Internals

Even though Ghidra has a rather extensive set of features built-in, the nature of software reverse engineering requires one to dive into edge cases. Rather than waiting for anyone to create a built-in feature in Ghidra to solve such an issue, the framework offers an extensively documented API to write scripts for. These scripts are natively written in Java, but can also be written in Python 2, which are then executed via Jython. Support for Python 3 is also possible, when using third party extensions. In this case, the focus is on Java, as this is Ghidra’s native tongue.

The Ghidra script for SHAREM (dubbed “Sharem.java”) contains several steps, outlined in the flow chart below.

https://t.me/learningnets

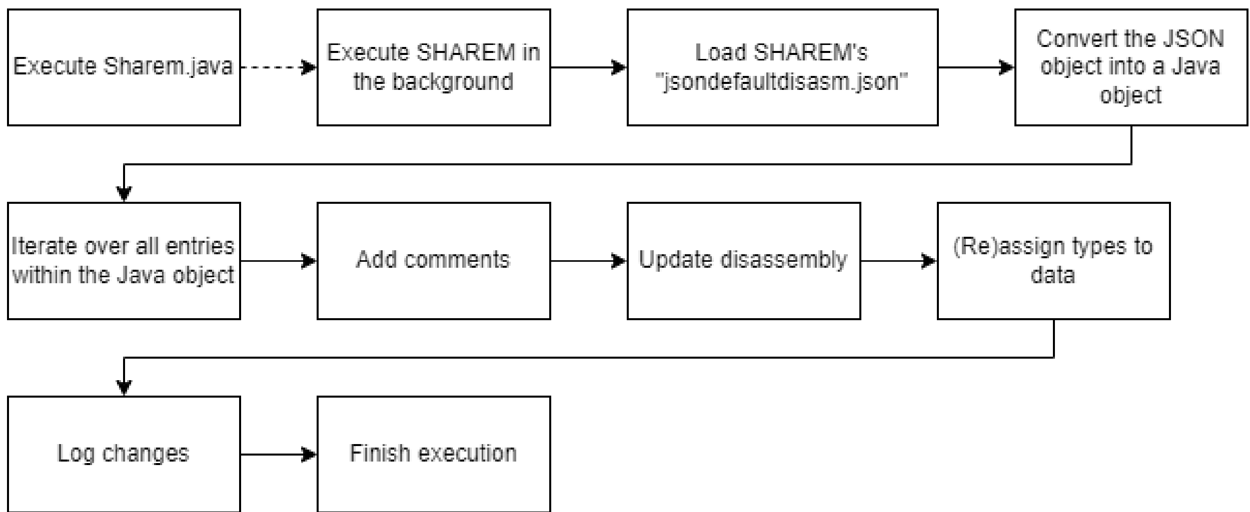

Fig. 3. SHAREM's JSON goes through various trasnformational stages after being loaded into Ghidra.

In short, the JSON output from SHAREM is loaded, and parsed into Java objects. These Java objects are then iterated, allowing Ghidra to add comments to given offsets, update the disassembly by updating the bytes, and by assigning data types to data structures. Ghidra can, by design, execute this script when running headless itself. As such, it can be used to fully automate processes, since both SHAREM and Ghidra can run headless.

The code itself is explained with the help of several figures below, excluding helper functions. At first, SHAREM is executed, and the path to the shellcode needs to be provided. One could make these arguments to this script as well, meaning they can be passed via the command-line interface when running Ghidra headless. It also assumes SHAREM is configured to run headless prior to execution.

```java
@Override
protected void run() throws Exception {
    // The directory in which the command is executed
    File workingDirectory = new File("C:\\sharem");
    // The command to execute within said working directory
    String command = "C:\\Python311\\python.exe main.py -r32 C:\\shellcode.bin";


    try {
        /*
         * Executes the given command from within the given working directory. This
         * function only returns once the command has been executed.
         */
        execute(workingDirectory, command);
    } catch (Exception ex) {
        // Print the error message
        printerr(ex.getMessage());
        // Return early
        return;
    }

    // Instantiate a new Gson object for later use
    Gson gson = new Gson();

    // Declare the JSON file variable, based on the working directory
    File jsonFile = new File(
            workingDirectory.getAbsolutePath() + "\\sharem\\sharem\\sharem\\logs\\default\\jsondefaultdisasm.json");

    // Read the file and store the result in a string
    String json = Files.readString(jsonFile.toPath());

    // Convert the raw JSON into a Java object
    SharemObject result = gson.fromJson(json, SharemObject.class);
```

Fig. 4. A snippet of code showing the Ghidra code to utilize SHAREM output.

https://t.me/learningnets

On line 59, the JSON output from SHAREM is read, which is converted into Java objects by Google's GSON library. The "SharemObject" class is present within "Sharem.java" and contains fields with the same names and structure as the JSON file that has previously been outlined.

Next, all objects are iterated over, and handled depending on their properties. If the comment is not empty, it is set at the given address, and a message is printed in Ghidra's scripting console.

```
60
61         // Iterate over the objects
62         for (SharemSubObject object : result.getObjects()) {
63             // If a given object's comment is not null, empty, nor white space
64             if (object.getComment().isBlank() == false) {
65                 // Get the offset in hexadecimal format
66                 long offset = Long.parseLong(object.getAddress().substring(2), 16);
67                 // Set a comment at the given offset, with the given comment
68                 setPreComment(toAddr(offset), object.getComment());
69                 // Create a string to print debug information
70                 String message = "Commented \"" + object.getComment() + "\" at " + object.getAddress();
71                 // Print the message
72                 println(message);
73             }
74
75             // If the type is CODE
76             if (object.getBytes().equalsIgnoreCase("CODE")) {
77                 // Get the value of the bytes
78                 byte[] sharemValues = getBytesFromSharemObject(object);
79                 if (sharemValues == null) {
80                     continue;
81                 }
82                 // Get the bytes from Ghidra's listing
83                 byte[] ghidraValues = getBytes(toAddr(object.getAddress()), object.getSize());
84                 // If the values aren't equal
85                 if (Arrays.compare(sharemValues, ghidraValues) > 0) {
86                     // Clear the listing
87                     clearListing(toAddr(object.getAddress()));
88                     // Set the bytes as provided by SHAREM
89                     setBytes(toAddr(object.getAddress()), sharemValues);
90                     // Disassemble the bytes
91                     disassemble(toAddr(object.getAddress()));
92                 }
```

Fig. 5. The Ghidra plugin iterate s over different objects, handling each differently based on their properties.

If the current object's bytes equal "CODE", the bytes within Ghidra are compared to the ones in the output from SHAREM. If SHAREM's bytes are different, they are replaced, and they are disassembled by Ghidra.

https://t.me/learningnets

```
93              } else if (object.getBytes().equalsIgnoreCase("DATA")) { // If the type is DATA
94                  if (object.getDataType().equalsIgnoreCase("String")) { // if the type is a string
95                      // Get the data at the given address
96                      Data data = getDataAt(toAddr(object.getAddress()));
97                      // If the data is not null
98                      if (data != null) {
99                          // Compare the length of the data and the size mentioned in the SHAREM object
100                         if (data.getLength() != object.getSize()) {
101                             // Get the end address
102                             Address end = toAddr(object.getAddress()).add(toAddr(object.getSize()).getOffset() - 1);
103                             // Get the start address
104                             Address start = toAddr(object.getAddress());
105                             // Clear the listing
106                             clearListing(start, end);
107                             // Create a string at the given address
108                             createAsciiString(toAddr(object.getAddress()));
109                         }
110                     } else {
111                         // Get the end address
112                         Address end = toAddr(object.getAddress()).add(toAddr(object.getSize()).getOffset() - 1);
113                         // Get the start address
114                         Address start = toAddr(object.getAddress());
115                         // Clear the listing
116                         clearListing(start, end);
117                         // If no data is present, simply create a string at the address
118                         createData(toAddr(object.getAddress()), StringDataType.dataType);
119                     }
120                 } else if (object.getDataType().equalsIgnoreCase("API Pointer")) { // If the type is a pointer
121                     // Get the data at the address
122                     Data data = getDataAt(toAddr(object.getAddress()));
123                     // If the data is not present
124                     if (data == null) {
125                         // Create the pointer
126                         createData(toAddr(object.getAddress()), PointerDataType.dataType);
127                     }
128                     // Set a comment with the instruction to provide context
129                     setPreComment(toAddr(object.getAddress()), object.getInstruction());
```

Fig. 6. The Ghidra plugin offers several different ways of handling different types of data.

If the type is "DATA", several sub sections are possible, where the type is a string, API pointer, or no specified type. The type itself is created, and comments are set. If no type is specified, the data will be assigned a type purely based on its size, allowing the analyst to easily see the size of accessed data's size. This allows an analyst to more easily recognise checksums, constants, and other values that one might encounter.

5.2 Using the Ghidra script

Armed with an understanding of the script, knowing how to execute said script is crucial to put it to good use. While Ghidra's CodeBrowser is open, the green play button in the icon row opens the Script Manager, as can be seen in the figure below.

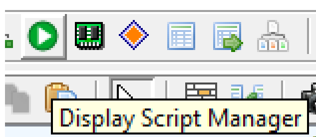

Fig. 7. CodeBrower can be used to access the Script Manager in Ghidra

Alternatively, one can open the Window tool strip menu item, and select Script Manager. Once open, one can manage the script directories with the hamburger menu in the top right corner of the Script Manager.

https://t.me/learningnets

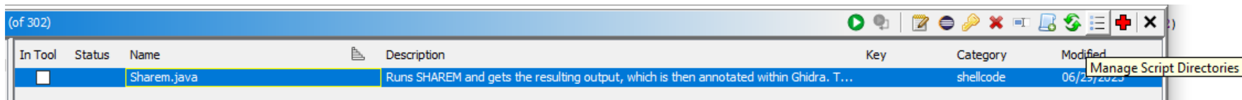

Fig. 8. The green plus can be used to add a folder to the list of locations where scripts are fetched from

Within that menu, one can click on the green plus to add a folder to the list of locations where scripts are fetched from. Once added, press the two green arrows to refresh the list.

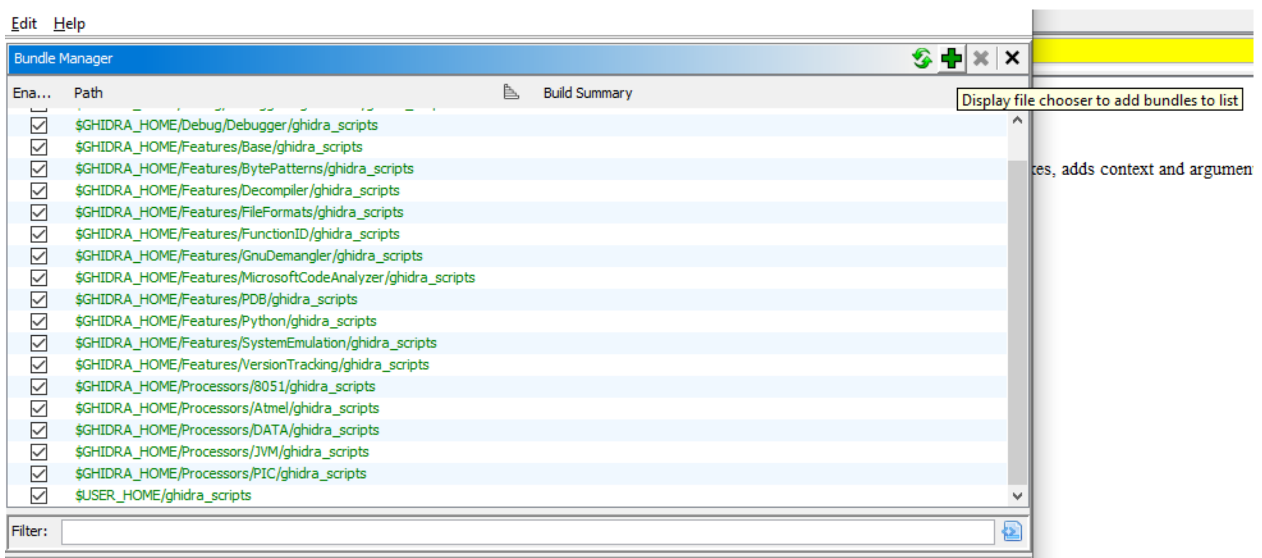

Fig. 9. Once the script is added to the list, it can now be searched for using the textbox next to the filter label.

Once the refresh action has completed, the script should be in the list, and can be searched for using the textbox next to the filter label, as seen in figure 21. To execute a script, simply double click it, or press the green play button within the Script Manager.

The Script's Output

With an understanding of how to ingest SHAREM's output, how the Ghidra script works, and how to execute said Ghidra script, it is time to show the outcome of the script for a given sample. The two screenshots below show the same piece of code, where the first is based on Ghidra's original analysis, the second has been annotated with comments taken from SHAREM.

```
Decompile: FUN_00000005 - (na.bin)
17  puVar10 = &stack0x00000004;
18  param_2 = param_2 + -5;
19  FUN_0000007a();
20  FUN_000000b0(param_2,&stack0x00000004);
21  iVar6 = param_2 + 0x144;
22  iVar4 = (**(code **)(param_2 + 0x12d))(iVar6,param_2,puVar10);
23  iVar2 = iVar6 + 0x131;
24  iVar9 = iVar2;
25  uVar5 = (**(code **)(iVar6 + 0x125))(iVar4,iVar2,iVar6,param_2);
26  *(undefined4 *)(iVar4 + 0x14f) = uVar5;
27  (**(code **)(iVar4 + 0x14f))(0,iVar4 + 0x15f,iVar4 + 0x153,0,0,iVar4,iVar9);
```

https://t.me/learningnets

Fig. 10. This representative portion of a shellcode has been analyzed by Ghidra, but is unable to identify any WinAPIs.

```
Decompile: FUN_00000005 - (na.bin)
19                    /* ; GetPC */
20  puVar11 = &stack0x00000004;
21  iVar6 = unaff_retaddr + -5;
22  FUN_0000007a();
23  FUN_000000b0(iVar6,&stack0x00000004);
24  iVar10 = iVar6 + 0x144;
25                    /* ; call to LoadLibraryA
26                             (urlmon.dll) */
27  iVar4 = (**(code **)(iVar6 + 0x12d))(iVar10,iVar6,puVar11);
28  iVar2 = iVar10 + 0x131;
29                    /* ; call to GetProcAddress
30                             (urlmon.dll, URLDownloadToFileA) */
31  iVar9 = iVar2;
32  uVar5 = (**(code **)(iVar10 + 0x125))(iVar4,iVar2,iVar10,iVar6);
33  *(undefined4 *)(iVar4 + 0x14f) = uVar5;
34                    /* ; call to URLDownloadToFileA
35                             (0x0, http://127.0.0.1/file.exe,
36                              C:\file.exe, 0x0, 0x0) */
37  (**(code **)(iVar4 + 0x14f))(0,iVar4 + 0x15f,iVar4 + 0x153,0,0,iVar4,iVar9);
```

Fig. 11. Once the SHAREM output has been ingested, a tremendous amount of detail has been added to the analysis results.

Code is important with shellcode, but equally important is data. Needless to say, not all data is of an identical type. With regards to data types, the figure below shows the before and after on the left and right respectively. This allows an analyst to immediately see the used values, and clearly differentiates between pointers, strings, and constants. These distinctions immediately add clarity to the analysis.

```
0000011f ac          ??     ACh
00000120 c8          ??     C8h
00000121 ff          ??     FFh
00000122 ff          ??     FFh
00000123 00          ??     00h
00000124 00          ??     00h
00000125 01          ??     01h
00000126 00          ??     00h
00000127 00          ??     00h
00000128 00          ??     00h
00000129 02          ??     02h
0000012a 00          ??     00h
0000012b 00          ??     00h
0000012c 00          ??     00h
0000012d 03          ??     03h
0000012e 00          ??     00h
0000012f 00          ??     00h
00000130 00          ??     00h
00000131 55 52 4c    ds     "URLDownloadToFileA"
         44 6f 77
         6e 6c 6f ...
00000144 75 72 6c    ds     "urlmon.dll"
         6d 6f 6e
         2e 64 6c ...
00000150 00          ??     00h
00000151 00          ??     00h
00000152 00          ??     00h
00000153 43 3a 5c    ds     "C:\\file.exe"
         66 69 6c
         65 2e 65 ...
0000015f 68 74 74    ds     "http://127.0.0.1/file.exe"
```

```
00000115 ee ea c0 1f    ddw     1FC0EAEEh
00000119 4a 37 a1 49    ddw     49A1374Ah
0000011d 26 80 ac c8    ddw     C8AC8026h
00000121 ff ff 00 00    ddw     FFFFh
                 GetProcAddress - API pointer
00000125 01 00 00 00    addr    LAB_00000000+1
00000129 02 00 00 00    ddw     2h
                 LoadLibraryA - API pointer
0000012d 03 00 00 00    addr    LAB_00000000+3
00000131 55 52 4c       ds      "URLDownloadToFileA"
         44 6f 77
         6e 6c 6f ...
00000144 75 72 6c       ds      "urlmon.dll"
         6d 6f 6e
         2e 64 6c ...
                 URLDownloadToFileA - API pointer
0000014f 00 00 00 00    addr    00000000
00000153 43 3a 5c       ds      "C:\\file.exe"
         66 69 6c
         65 2e 65 ...
0000015f 68 74 74       ds      "http://127.0.0.1/file.exe"
         70 3a 2f
         2f 31 32 ...
```

Fig. 12. With SHAREM's output ingested by Ghidra, we can identify several API pointers, as shown.

Additionally, the Ghidra Plugin provides benefit to Ghidra's Console after executing the script with the SHAREM emulation results. The hexadecimal values for offsets are clickable, after which the disassembly and decompiler views will align with said offset, as seen in the figure below. This allows for easy navigation to interesting artifacts based on the log.

https://t.me/learningnets

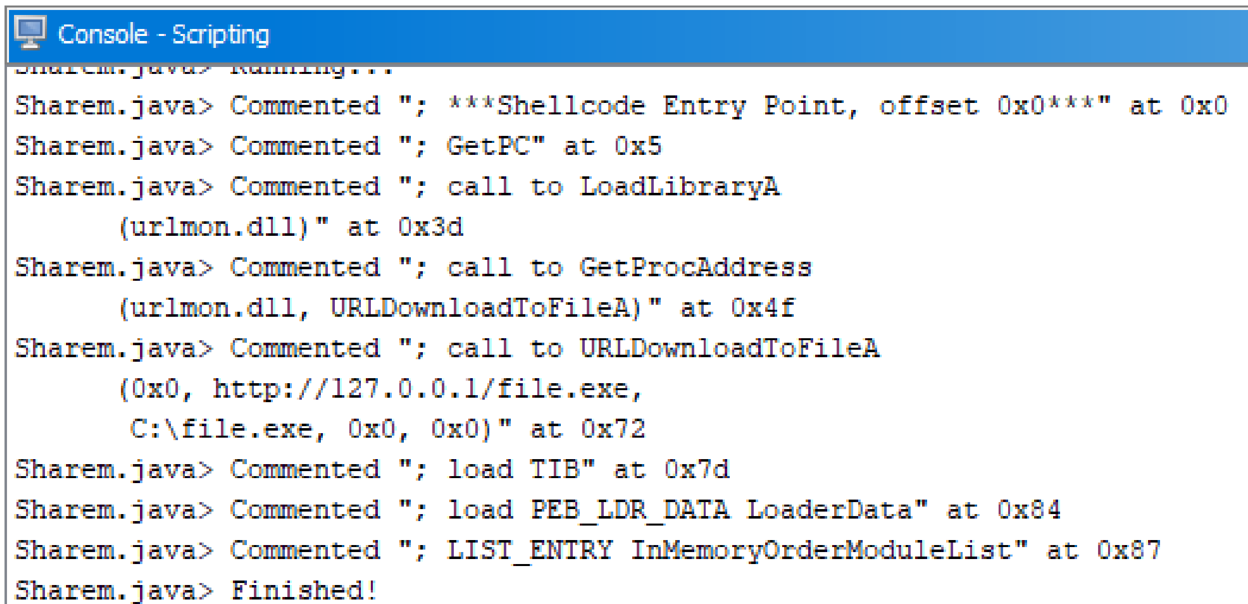

Fig. 13. The hexadecimal values in the console output are clickable, allowing easy navigation to key parts of the disassembly.

6 Validation

SHAREM has made significant contributions with respect to shellcode analysis. The central purpose of the framework was to be able to provide in-depth analysis of shellcode samples, providing novel contributions in the areas of emulation and disassembly. The final step of that research was to implement SHAREM and benchmark it against various samples, thereby providing validation through demonstrating its efficacy. As part of this research, we obtained numerous modern Windows shellcode samples from various public and private sources, so as to be able to test SHAREM. Additionally, some samples were taken from recent malware samples via CAPE Sandbox, while both using and not using PE-Sieve [23]. Finally, we also created a set of samples using PE_to_Shellcode [24], which was able to convert PE files into actual shellcode. This was done to more than 200 normal PE files as well as malware. Some samples were not utilized, such as those that used WinAPI functions with hardcoded addresses. Those were created before the advent of ASLR, and they were relics that would no longer work in any modern sample. Those would not be supported on SHAREM, although we could if we wanted to; it just would not be desirable to do so. Additionally, several hundred testing samples were created, to ensure that all custom implementations of WinAPIs and Windows syscalls behaved correctly.

Additionally, SHAREM provides multiple unprecedented, novel contributions. Each of these was rigorously tested with dozens or in some cases hundreds of test cases. In all, thousands of tests have been conducted on many aspects of SHAREM. A more formal academic paper will discuss these more in depth at a later date, but briefly we will mention the highlights. We have extensively tested SHAREM's ability to automatically decode obfuscated shellcode, allowing its decoded form to be recovered. In all cases that we tested, SHAREM was successful at recovering the decoded form of the shellcode, though with a caveat. Some data could be updated by the shellcode, which may not have existed originally. However, this only applies to a narrow subset of data present in shellcode samples. For instance, SHAREM may have a placeholder for the runtime addresses of API's. Once the shellcode can discover an API's runtime address, it will update this location with the newly found address. We do not regard this as a design flaw, but we note it nonetheless. SHAREM's ability to integrate emulation data into the disassembly has been tested extensively. While there are occasionally some minor issues, mostly it is around 99% accurate for shellcode that has been emulated. In general, with respect to the disassembly of shellcode, we can regard SHAREM as being significantly more accurate than the leading disassemblers.

https://t.me/learningnets

7 Contributions

SHAREM provides significant contributions. SHAREM contains a novel shellcode emulator, which can log more than 20,000 WinAPI's and virtually all user-mode Windows syscalls. The latter had never before been emulated, and in fact the preparation to allow SHAREM to be able to successfully emulate Windows syscalls was so extensive that it led to a new offensive security tool, ShellWasp, which formally introduced a new approach to shellcode that utilizes Windows syscalls – that is, syscall shellcode. This was due to the fact that prior to our research, there have been virtually no instances of shellcode that used Windows syscalls, aside from egghunters. SHAREM's complete code coverage is an important contribution, allowing it to generate complete emulation data that can be utilized to enhance the disassembly and to discover more additional functionality. In fact, complete code coverage can be regarded as a new area of research. SHAREM can successfully deobfuscate an encoded shellcode, allowing for its original form to be recovered, even if the author goes to lengths to prevent this, such as by reencoding the previously decoded shellcode. SHAREM can provide the disassembly of the decoded shellcode, immediately after emulation, blurring the lines between this disassembler and debugger. Finally, SHAREM is the first tool to provide timeless debugging capabilities specifically for shellcode.

8 Conclusions

SHAREM is a game changer with respect to the analysis of shellcode, particularly encoded shellcode. Now an individual can take a shellcode that may be found as part of incident response, and potentially be able to easily and quickly defeat the shellcode's encoding to not only reveal its functionality, but also to see the original Assembly of the shellcode, complete with all API's labeled.

Additionally, while the purview of SHAREM is shellcode, it has novel contributions that likely will spread to other areas. The ability to perform complete code coverage is game changing. SHAREM with obfuscated shellcode can blur disassembly and debugging, creating a special enhanced disassembly. And, not only is SHAREM able to act as a standalone tool for the security researcher or malware analyst, but it also can be used and deployed externally for web sites or other systems where samples can be submitted for analysis, as it generates JSONs, which can be ingested by web services, with any results found by SHAREM being able to be displayed—including an extensive reporting of attributes, as well as even the disassembly itself.

SHAREM will serve as the definitive shellcode analysis framework, providing many unique capabilities for both malware analysts and security researchers.

References

1. Brizendine, B.: ShellWasp, https://github.com/Bw3ll/ShellWasp
2. Zimmer, D.: Scdbg Shellcode Analysis, http://sandsprite.com/CodeStuff/scdbg_manual/MANUAL_EN.html
3. FireEye: Speakeasy, https://github.com/fireeye/speakeasy
4. Borders, K., Prakash, A., Zielinski, M.: Spector: Automatically analyzing shellcode. Proc. - Annu. Comput. Secur. Appl. Conf. ACSAC. 501–514 (2007). https://doi.org/10.1109/ACSAC.2007.11
5. Fratantonio, Y., Kruegel, C., Vigna, G.: Shellzer: a tool for the dynamic analysis of malicious shellcode. In: International workshop on recent advances in intrusion detection. pp. 61–80 (2011)
6. Bayer, U., Moser, A., Kruegel, C., Kirda, E.: Dynamic analysis of malicious code. J. Comput. Virol. 2, 67–77 (2006)
7. Akritidis, P., Markatos, E.P., Polychronakis, M., Anagnostakis, K.: Stride: Polymorphic sled detection through instruction sequence analysis. IFIP Adv. Inf. Commun. Technol. 181, 375–391 (2005). https://doi.org/10.1007/0-387-25660-1_25
8. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. Proc. 12th USENIX Secur. Symp. 169–186 (2003)
9. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: 2005 IEEE Symposium on Security and Privacy (S&P'05). pp. 32–46 (2005)
10. Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Network--level polymorphic shellcode detection using emulation. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 54–73 (2006)
11. Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Emulation-based detection of non-self-contained

https://t.me/learningnets

polymorphic shellcode. In: International Workshop on Recent Advances in Intrusion Detection. pp. 87–106 (2007)

12. Chinchani, R., Van Den Berg, E.: A fast static analysis approach to detect exploit code inside network flows. In: International Workshop on Recent Advances in Intrusion Detection. pp. 284–308 (2005)
13. Brumley, D., Newsome, J., Song, D., Wang, H., Jha, S.: Towards automatic generation of vulnerability-based signatures. In: 2006 IEEE Symposium on Security and Privacy (S&P'06). pp. 15--pp (2006)
14. Newsome, J., Karp, B., Song, D.: Polygraph: Automatically generating signatures for polymorphic worms. In: 2005 IEEE Symposium on Security and Privacy (S&P'05). pp. 226–241 (2005)
15. Payer, U., Teufl, P., Lamberger, M.: Hybrid engine for polymorphic shellcode detection. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 19–31 (2005)
16. Detristan, T., Ulenspiegel, T., Malcom, Y., Underduk, M.: Polymorphic shellcode engine using spectrum analysis, (2003)
17. Wang, X., Pan, C.-C., Liu, P., Zhu, S.: Sigfree: A signature-free buffer overflow attack blocker. IEEE Trans. dependable Secur. Comput. 7, 65–79 (2008)
18. Van Eeckhoutte, P.: Windows 10 Egghunter (Wow64) and More, https://www.corelan.be/index.php/2019/04/23/Windows-10-egghunter/
19. Stevens, D.: Hancitor Maldoc Bypasses Application Whitelisting, https://isc.sans.edu/forums/diary/Hancitor+Maldoc+Bypasses+Application+Whitelisting/21683/
20. Bania, P.: Windows Syscall Shellcode, http://piotrbania.com/all/articles/Windows_syscall_shellcode.pdf
21. Jurczyk, M.: Windows X86-64 System Call Table (XP/2003/Vista/2008/7/2012/8/10), https://j00ru.vexillium.org/syscalls/nt/64/
22. Hotz, G.: Timeless debugging. (2016)
23. Doniec, A.: PE-Sieve, https://github.com/hasherezade/pe-sieve
24. Doniec, A.: PE to Shellcode, https://github.com/hasherezade/pe_to_shellcode

https://t.me/learningnets