2022 PRAGUE

28 - 30 September, 2022 / Prague, Czech Republic

SHAREM: SHELLCODE ANALYSIS FRAMEWORK WITH EMULATION, A DISASSEMBLER, AND TIMELESS DEBUGGING

Bramwell Brizendine, Jacob Hince, Austin Babcock, Tarek Abdelmotaleb, Sascha Walker & Shelby VandenHoek

VERONA Lab, USA

bramwell.brizendine@gmail.com
jacob.hince@trojans.dsu.edu
austin.babcock@trojans.dsu.edu
tarek.abdelmotaleb@trojans.dsu.edu
sascha.walker@trojans.dsu.edu
shelby.vandenhoek@trojans.dsu.edu

ABSTRACT

SHAREM is a state-of-the-art framework to analyse and emulate shellcode. SHAREM's emulator can handle more than 16,000 *Windows* APIs, and is the first tool to emulate *Windows* syscalls. It additionally features its own disassembler, which is significantly more accurate than leading disassemblers for shellcode. The disassembler can also pair emulation data with disassembly results to produce highly accurate shellcode, with all function calls identified. Uniquely, SHAREM can take an encoded shellcode and present its decoded form in the disassembler, allowing for more meaningful analysis. It also supports identification of many unique shellcode attributes, and it supports brute-force deobfuscation without emulation.

1. INTRODUCTION

Random shellcode found online or extracted from malware can be deeply enigmatic; comprehending its functionality is not straightforward. SHAREM is a cutting-edge shellcode analysis framework that provides many powerful, unique capabilities to unravel the mysteries of shellcode.

SHAREM provides novel contributions to solve several problems regarding shellcode. This involves emulation of highly complex *Windows* shellcode, utilizing both WinAPI functions and syscalls. *Windows* shellcode uses esoteric techniques to covertly load and use many WinAPI functions. These APIs are often passed as checksums, thwarting static analysis. However, SHAREM can emulate highly complex, encoded *Windows* shellcode, identifying over 16,000 APIs, parameters, and return values. Additionally, over 550 *Windows* syscalls called in shellcode can be emulated with parameters enumerated.

Analysis of modern *Windows* shellcode can be problematic with respect to disassembly, and disassemblers often produce wildly inaccurate disassembly. SHAREM features its own disassembler, which creates disassembly with up to 96% accuracy; the disassembler can use emulation to enhance its accuracy further. The disassembly is richly annotated, indicating WinAPIs and syscalls called, alongside their parameters. SHAREM's handling of encoded shellcode allows the decrypted form to be viewable immediately in the disassembler, as if it were not encrypted.

1.1. WinAPIs in shellcode

Shellcode has become pervasive in the malware ecosystem, as malware often utilizes shellcode to hinder analysis efforts. In exploitation, shellcode can be effective at covertly loading malicious functionality. With the prevalence of ASLR, by necessity all modern *Windows* shellcode must utilize introspection techniques to discover the addresses of both DLLs and WinAPI functions. This introspective journey begins by walking the Process Environment Block (PEB) and finding the PEB_LDR_DATA. The PEB is pointed to by FS:0x30 or GS:0x60, depending on architecture, making it easily accessible with simple Assembly expressions. At the PEB_LDR_DATA we find three module lists, beginning with InLoadOrderModuleList. One of these doubly linked lists can be explored further to eventually discover the base address of a DLL. The base address also happens to be the same location as the start of a PE file. Thus, different structures and fields within the PE file format can be traversed, until the exports table is found. Then the APIs can be looped through until the desired WinAPI is found. As part of a loop, the name of the WinAPI can be checked for directly, though frequently a checksum is utilized, rather than its plaintext name. If checksums are used, the shellcode hashes all names within the exports table until it finds a match with the desired checksum. As checksums are one-way hashes, this enhances the difficulty of understanding the shellcode, as string names need not be in the shellcode. Once the WinAPI is found, its runtime address can be found in a parallel array, and the address can be saved for future use. Shellcode can be modular, with many WinAPIs called from multiple DLLs. The only limitation is the size of the shellcode, which can be an issue in exploitation, though not necessarily with malware.

1.2 Syscalls in shellcode

While very rare in shellcode form, *Windows* syscalls have recently become popular and trendy in both red-team software and malware. As part of our research, we created pure shellcode with syscalls for both *Windows 7* and *Windows 10*. While syscalls have been used for many years as shellcode in the form of egghunters, they have been extremely rare in shellcode form for purposes other than egghunting. Syscalls are relatively simple to use in higher level languages, but they are significantly challenging to create as shellcode to be injected into memory. While higher level languages often use syscalls in Assembly form, it is strikingly different from position-independent shellcode, as the higher-level language can set up resources needed for syscalls. NTDLL functions can be reverse engineered to reveal the System Service Number (SSN) of syscalls. *Windows* syscalls transition from user mode to kernel mode, allowing system functionality to be invoked from user mode [1]. Syscalls are attractive as they protect against hooking.

In 32-bit applications, syscalls are called by using a function pointed to by offset 0xC0 of the TEB, or FS:0xc0; in *Windows 10*, an offset in NTDLL is used to initiate syscalls, although fs:0xc0 is maintained for backwards computability. In 64-bit *Windows*, the *syscall* instruction is used. Syscalls are invoked by placing the SSN into *eax* and pushing necessary parameters before calling the syscall. We reverse engineered *Windows 10* and *Windows 7* to discover precisely how to implement syscalls in SHAREM, enabling us to build shellcode with syscalls for both. (We discovered that *Windows 7* shellcode must be specially modified to work in *Windows 10*.)

2. RELATED WORK

In the academic literature, Spector [2] is a POC shellcode emulator that utilized symbolic execution to extract WinAPIs and parameters, simulating appropriate responses for some WinAPIs. This unreleased tool supported only 23 WinAPIs. Another unreleased academic tool was Shellzer [3], supporting a limited number of API functions. Rather than emulating shellcode, Shellzer actually executed and single-stepped through the shellcode with the PyDbg library. To find WinAPIs, Shellzer would debug and check if the program counter pointed to a *Windows* DLL; if so, it would determine if it corresponded to a WinAPI. Although not an emulator, Shellzer could enumerate a limited subset of WinAPIs.

There are two primary industry tools to analyse shellcode, Scdbg [4] and SpeakEasy [5]. Scdbg is a well-known, older tool for shellcode emulation, supporting 245 opcodes, 15 DLLs and 242 APIs. Scdbg is effective for its supported subset of APIs. SpeakEasy is a recent framework for emulating user- and kernel-mode malware; shellcode is also supported, although it is not the tool's focus. SpeakEasy was released after our NSA grant funding was obtained for SHAREM and after development commenced. We have not examined SpeakEasy closely, and it has not influenced SHAREM. SpeakEasy supports both 32- and 64-bit shellcode. While Scdbg and SpeakEasy are impressive tools, we have found failure points with more complex and advanced shellcode samples, which can be handled by SHAREM. In general, we have found SHAREM successfully able to emulate and discover significantly more WinAPIs than either. Additionally, a cursory search on *GitHub* reveals several small, personal projects with limited shellcode emulation; these seem incomplete and support just a handful of functions.

3. SHAREM ARCHITECTURE

SHAREM's architecture consists of several components. The most important is the emulator, supporting up to 16,000 WinAPIs and 550 *Windows* syscalls. Next, the disassembly analysis engine produces disassembly through custom analysis phases; it can also pair that with emulation results, for highly accurate disassembly. Next, the shellcode instruction detection engine discovers several instructions or techniques related to shellcode or malware, providing intelligence that can be reported on and used as comments in disassembly. The strings discovery engine has original detection for ASCII, Unicode, and Push stack strings; this also influences disassembly analysis. Finally, the brute-force deobfuscation component decrypts shellcode samples without using emulation; it supports parallel processing and distributed computing. In fact, the brute-force deobfuscation works even if no decoder stub is present.

4. SHAREM EMULATOR

Emulation serves many roles. The most important role of the emulator is to discover WinAPIs and *Windows* syscalls. In fact, SHAREM is the only tool that emulates *Windows* syscalls. The emulation is also used to enhance the disassembly generated. Finally, the emulator deobfuscates encoded shellcode.

Unicorn-Engine

SHAREM utilizes Unicorn-Engine to provide emulation support. Unicorn-Engine provides a virtual memory interface from which to write and read, and SHAREM builds on top of the emulation everything required to support emulation of *Windows* shellcode. There is a very significant, non-trivial amount of work that must be done in order for *Windows* shellcode to successfully emulate in our environment.

4.1 Initial setup of SHAREM

SHAREM copies the required DLLs from SysWow64 and System32 directories, inflating each the necessary amount. Most DLLs are naturally inflated when going from file to process memory, so each is inflated just the right amount, according to PE file attributes. When emulated, these are mapped to process memory, and many must be inflated. The pefile Python library then parses each DLL, identifying the function addresses. This step is completed only once in *Windows*, taking several minutes. For *Linux*, an alternative setup will be available, using already extracted and parsed DLLs.

4.2 Function emulation

For WinAPI emulation, we must implement several *Windows* internal structures. This includes setting up the PEB, the TEB, the PEB LDR, and the module doubly linked lists. Each must be populated with correct and appropriate values, to support both 32- and 64-bit architectures. Additionally, we parse and modify *Windows* system DLLs, before loading those into memory. Presently, 31 DLLs are supported with approximately 16,000 *Windows* APIs. Since some shellcode behaves unusually when traversing PE files, loading these into memory, rather than simply emulating some of their attributes, allows for optimum behaviour. If we just simulate aspects of these, complex shellcode may cause emulation to fail.

4.2.1 Lookup dictionaries for functions

For both WinAPIs and syscalls, original, custom lookup dictionaries are used for DLLs, providing detailed function prototype information. This allows for SHAREM to support thousands of APIs, and for each to be emulated correctly.

```
0x12000089 URLDownloadToFileA(LPUNKNOWN pCaller, LPCSTR szURL, LPCSTR szFileName, DWORD dwReserved,
LPBINDSTATUSCALLBACK lpfnCB)
        LPUNKNOWN pCaller: 0x0
        LPCSTR szURL: http://167.99.229.113/default.css
        LPCSTR szFileName: test.bat
        DWORD dwReserved: 0x0
        LPBINDSTATUSCALLBACK lpfnCB: 0x0
        Return: HRESULT S_OK

0x120001a9 VirtualAlloc(LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect)
        LPVOID lpAddress: 0x0
        SIZE_T dwSize: 0x1000
        DWORD flAllocationType: MEM_COMMIT
        DWORD flProtect: PAGE_EXECUTE_READWRITE
        Return: INT 0x25000000

0x12000190 CreateProcessA(LPCSTR lpApplicationName, LPSTR lpCommandLine, LPSECURITY_ATTRIBUTES
lpProcessAttributes, LPSECURITY_ATTRIBUTES lpThreadAttributes, BOOL bInheritHandles,
DWORD dwCreationFlags, LPVOID lpEnvironment, LPCSTR lpCurrentDirectory, LPSTARTUPINFO
lpStartupInfo, LPPROCESS_INFORMATION lpProcessInformation)
        LPCSTR lpApplicationName: [NULL]
        LPSTR lpCommandLine: cmd.exe /c test.bat
        LPSECURITY_ATTRIBUTES lpProcessAttributes: 0x0
        LPSECURITY_ATTRIBUTES lpThreadAttributes: 0x0
        BOOL bInheritHandles: FALSE
        DWORD dwCreationFlags: 0x0
        LPVOID lpEnvironment: 0x0
        LPCSTR lpCurrentDirectory: [NULL]
        LPSTARTUPINFO lpStartupInfo: 0x25000000
        LPPROCESS_INFORMATION lpProcessInformation:
                HANDLE hProcess: 0x88880000
                HANDLE hThread: 0x88880008
                DWORD dwProcessId: 0x2710
                DWORD dwThreadId: 0x4e20
        Return: BOOL TRUE
```

Figure 1: When showing emulation results, SHAREM locates human-readable equivalents to hexadecimal results. SHAREM additionally displays some structures, as seen with lpProcessInformation.

When shellcode traverses the DLLs in memory to find a runtime address, SHAREM looks up the address to see what function it corresponds to. Once the function name is found, another lookup can be made into our WinAPI and syscall dictionaries, to identify function prototype information for the API. This tells SHAREM how to process the function call.

```
************* APIs *************

0x1200003d LoadLibraryA(LPCTSTR lpLibFileName)
        LPCTSTR lpLibFileName: urlmon.dll
        Return: HMODULE 0x1554341c

0x12000073 URLDownloadToFileA(LPUNKNOWN pCaller, LPCSTR szURL, LPCSTR szFileName, DWORD dwReserved,
LPBINDSTATUSCALLBACK lpfnCB)
        LPUNKNOWN pCaller: 0x0
        LPCSTR szURL: http://www.fakedns.it/friend.hta
        LPCSTR szFileName: C:\Users\Public\friend.hta
        DWORD dwReserved: 0x0
        LPBINDSTATUSCALLBACK lpfnCB: 0x0
        Return: HRESULT S_OK

0x12000087 WinExec(LPCSTR lpCmdLine, UINT uCmdShow)
        LPCSTR lpCmdLine: mshta file://C:\Users\Public\friend.hta
        UINT uCmdShow: SW_HIDE
        Return: INT 0x20
```

Figure 2: SHAREM displays three APIs and parameters.

4.2.2 Logging Windows APIs and syscalls

Each time any call is made, it is checked to see if the destination matches a function. If so, then SHAREM logs the parameter information, return value, and site address of the call. SHAREM records each parameter according to its

indicated type. If the type is a string pointer, SHAREM logs the string. Similarly, if there is a structure pointer, rather than recording the pointer, SHAREM logs the entire structure with all its members, recording both data type and its value. At present, only some security-relevant structures are supported. SHAREM's lookup dictionaries support thousands of APIs. However, SHAREM also has many custom implementations. For each API considered security-relevant or used for malicious purposes, it has been examined to see if anything can be added with a custom implementation. In some cases, nothing of value can be added beyond what the lookup dictionaries do. Some functions have other special actions performed. For instance, with VirtualAlloc, memory is created according to passed arguments. Additionally, rather than logging hexadecimal values, SHAREM records its human-readable equivalent. Thus, with VirtualAlloc, for flProtect, it logs PAGE_EXECUTE_READWRITE, instead of 0x40. SHAREM provides numerous lookup dictionaries within custom API implementations, to obtain human-readable equivalents.

4.2.3 Emulation of syscalls

Only about 550 user-mode syscalls exist, documented by the security community [6], although there is no place that aggregates all syscall function prototypes. The vast majority of syscall function prototypes are undocumented by *Microsoft*. [7] lists approximately 250 NTDLL functions, although only some have a corresponding syscall. For SHAREM to work, we obtained nearly all user-mode syscall function prototypes, from numerous sources. We anticipate syscalls being used more in shellcode. Thus, SHAREM provides a rare instance of a defender being one step ahead of the adversary.

SHAREM is the only tool to emulate *Windows* syscalls. In so doing, SHAREM initializes fs:0xc0 to point to a designated location. Thus, calls to this site will indicate a syscall being made by the shellcode, and it can be correctly emulated. It is logged according to the user-selected *Windows* platform. Because syscall SSNs change regularly, the specific OS release must be provided. SHAREM uses different lookup dictionaries based on the release. Once identified, that data is used in the syscall function prototype dictionary.

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

0x1200002a NtAllocateVirtualMemory(HANDLE ProcessHandle, PVOID *BaseAddress, ULONG_PTR ZeroBits, PSIZE_T
RegionSize, ULONG AllocationType, ULONG Protect)
        HANDLE ProcessHandle: 0x0
        PVOID *BaseAddress: 0x16fffff8 -> 0x25000000
        ULONG_PTR ZeroBits: 0x0
        PSIZE_T RegionSize: 0x16ffffc4 -> 0x4000
        ULONG AllocationType: MEM_COMMIT | MEM_RESERVE
        ULONG Protect: PAGE_EXECUTE_READWRITE
        Return: NTSTATUS STATUS_SUCCESS
        EAX:  0x15 - (Windows 7, SP SP1)
```

Figure 3: SHAREM displays the syscall NtAllocateVirtualMemory, a deeper version of VirtualAlloc.

4.2.4 Checksums

Often, shellcode uses cryptographic hashes rather than a plaintext WinAPI name. Because SHAREM correctly implements *Windows* internal structures and provides the actual PE files, all WinAPIs can be discovered, despite checksums being used.

4.2.5 Simulating function success

Sophisticated shellcode utilizes both output parameters and return values in subsequent functions. SHAREM's intent is always to simulate function success. Thus, SHAREM might log 'S_OK', while placing 0x0 in EAX. Return success is achieved differently for each function, helping ensure as much malicious functionality as possible is discovered, lest a shellcode prematurely terminate. As samples are emulated in isolation, some environmental resources for real-world execution may be absent. For instance, a resource to download may have been removed, so SHAREM simulates success. Additionally, for functions such as LdrLoadDll, the NTDLL equivalent of LoadLibrary, SHAREM populates output parameters, such as the PHANDLE ModuleHandle, giving the address of the loaded DLL. Subsequent WinAPIs may use this handle, so the library's emulated address is written there.

4.2.6 Breaking out of loops

Breaking out of loops is not a novel concept, but it is a necessary one for a good emulator. As such, SHAREM maintains optional support to break out of loops, allowing the user to specify the upper threshold.

4.3 Capturing disassembly data for emulation

With each instruction executed, SHAREM records data that can be paired with its disassembly engine, to achieve more perfect disassembly. For each instruction executed, SHAREM records the offset it began, the size of the instruction, and it labels that byte as an instruction. Later, when paired with the disassembler, this metadata ensures those are bytes are classified as instructions, and that each begins at the correct offset.

4.4 Complete code coverage

One novel concept provided by SHAREM for shellcode emulation is complete code coverage. Inspired in part by evolutionary fuzzers, such as AFL, SHAREM can record all control flow paths taken and not taken, and each byte traversed is logged. If enabled, SHAREM will begin revisiting control flow paths not taken, after the emulation would have ended. SHAREM uses a novel approach of pushing objects for unvisited code paths onto a stack. Each object contains register CPU information along with stack values. Thus, the original CPU context can be restored, when revisiting the code path. Complete code coverage achieves two goals. Firstly, it ensures each instruction is executed, allowing for highly accurate disassembly. Secondly, SHAREM can discover additional functions that would be missed. In theory, this should allow all functionality to be discovered most of the time; one exception would be part of the shellcode not reachable by any code path.

4.5 Deobfuscation of encoded shellcode

Shellcode is often self-modifying code, allowing for encoded shellcode to be decrypted through emulation. Not only are functions enumerated, but SHAREM also uses unique functionality to recover the decrypted form of the shellcode. This is possible even if the shellcode decodes itself and then re-encodes those same bytes. SHAREM captures the decoded shellcode, sending it to the disassembler, as will be explored in the Disassembler section.

```
0xb3 call eax                              ff d0                 ..
    ;call to Sleep
      (0x3000)
0xb5 pop edx                               5a                    Z
0xb6 pop ebp                               5d                    ]
0xb7 push ebp                              55                    U
0xb8 push edx                              52                    R
0xb9 lea edi, [edx + 0x214]                8d ba 14 02 00 00     ......
0xbf push edi                              57                    W
0xc0 push dword ptr [edx + 0x1b3]          ff b2 b3 01 00 00     ......
0xc6 pop eax                               58                    X
0xc7 call eax                              ff d0                 ..
    ;call to DeleteFileA
      (test.bat)
```

Figure 4: Although this shellcode was heavily encoded, SHAREM presents its deobfuscated form in the disassembler, with functions and parameters identified.

4.6 Timeless debugging

SHAREM provides optional support for timeless debugging. When enabled, all CPU instructions executed are preserved, along with register values, both before and after. A minimalist approach, this secondary feature saves the above to a text file. Thus, an analyst could utilize timeless debugging to answer questions regarding the shellcode.

```
    >>> EAX: 0xf539a29c EBX: 0x1430cf18 ECX: 0x2eb EDX: 0x23116257 ESI: 0xc8ac8026 EDI: 0x14311ecb EBP: 0x1424b3b4 ESP: 0x16ffffe4
0x12000134  cmp byte ptr [edi], 0

    >>> EAX: 0xf539a29c EBX: 0x1430cf18 ECX: 0x2eb EDX: 0x23116257 ESI: 0xc8ac8026 EDI: 0x14311ecb EBP: 0x1424b3b4 ESP: 0x16ffffe4
0x12000137  jne 0x1200012e

    >>> EAX: 0xf539a29c EBX: 0x1430cf18 ECX: 0x2eb EDX: 0x23116257 ESI: 0xc8ac8026 EDI: 0x14311ecb EBP: 0x1424b3b4 ESP: 0x16ffffe4
0x1200012e  rol edx, 7
```

Figure 5: SHAREM's timeless debugging feature can let users look through hundreds of thousands of executed instructions, seeing how register values are impacted.

4.7 Handles and memory and management

SHAREM must manage the allocation of memory and handles. Many functions allocate memory or utilize allocated memory, and SHAREM Memory Manager (SMM) ensures memory is allocated at specific locations without collisions. Additionally, SHAREM Handle Manager (SHM) generates and maintains handles. Some may correspond to specific resources, such as a registry key, and SHM can log what the handle maps to. SHAREM can additionally track and display memory allocations, reads and writes.

4.8 SHAREM Registry Manager

The SHAREM Registry Manager (SMR) allows for values written to the registry to be used and retrieved subsequently; all registry WinAPIs are managed by SMR, and each is monitored for behaviours. Additionally, SMR allocates handles that map to registry key paths. SMR stores a list of registry values, and handles to key paths are used to retrieve registry values. Thus, when an HKEY handle is used, SHAREM logs the key itself and the handle ID. If a WinAPI attempts to read a

registry value not yet created, SMR simulates success, providing a dummy value; some dummy values are customizable via the config file.

```
*** Registry Actions ***
** Add **
\\RemoteComputer\HKEY_CLASSES_ROOT
HKEY_USERS\Software\Microsoft\Windows\CurrentVersion\RunServicesOnce
HKEY_LOCAL_MACHINE\Control Panel\Cursors

** Edit **
HKEY_USERS\Software
        OpenFirefox
        cmd \c C:\Program Files\Mozilla Firefox\firefox.exe

*** Registry Techniques ***
** Persistence **
HKEY_USERS\Software\Microsoft\Windows\CurrentVersion\RunServicesOnce

** Credentials **
\SECURITY\Policy\Secrets
```

Figure 6: SHAREM displays many registry actions as well as specific techniques used by the shellcode.

```
*** Registry Actions ***
** Add **
HKEY_CURRENT_USER\Software\Microsoft\Windows\DWM

** Edit **
HKEY_CURRENT_USER\Software\Microsoft\Windows\DWM
        503ce331-9dd0-4dad-ac4d-1e790569bac3a
        https://sharem.com/login/# :: Clipboard

*** Registry Hierarchy ***
** HKEY_Current_User **
HKEY_CURRENT_USER\Software\Microsoft\Windows\DWM
```

Figure 7: SHAREM can show what may be added or edited in the registry.

4.9 Emulation artifacts

Emulation allows numerous artifacts to be discovered. All API parameters are subjected to regular expressions, with values being saved to several categories and subcategories. Additionally, some values are taken directly from API parameters. For instance, parameters from all registry WinAPIs are logged into several registry sub-categories. SHAREM reports many diverse categories that may be interesting. Categories include command line instructions, URLs or web-related, registry files, executables, DLLs, and more.

```
************* Artifacts *************
*** Command Line ***
cmd.exe /c net user administrator test /active:yes
cmd.exe /c netsh advfirewall firewall add rule name="FW" dir=in action=allow protocol=TCP localport=27015

*** Web ***
0.0.0.0:27015
```

Figure 8: SHAREM captures many artifacts from shellcode emulation.

5. DISASSEMBLY ANALYSIS ENGINE

SHAREM's Disassembly Analysis Engine (DAE) was created due to how inaccurate disassembly of shellcode was in popular disassemblers, such as *IDA Pro*, *Binary Ninja* and *Ghidra*. While parts would be correct, each typically contained sections that were highly inaccurate, with data misclassified as instructions. Alternatively, disassembly would begin at incorrect offsets, causing incorrect disassembly to follow.

We have analysed shellcode samples with the original Assembly, comparing the disassembly produced by leading disassemblers with what it should be. Often, there were numerous mistakes. This is understandable as shellcode may be created manually, not following standard conventions, and intentionally difficult to follow. One of the goals with SHAREM is not just to have a token effort at disassembly, but to have disassembly that is actually close to correct.

To contend with the prevalence of inaccurate disassembly, SHAREM has multiple analysis phases, as described later. For each byte, it must be determined if it is data or instructions. In *x86 Intel* architecture, data and instructions are freely intermixed, and shellcode indeed has data in unusual locations. Data can include checksums or strings. Shellcode can have junk bytes inserted, and some code or data may be unreachable. Data may occur anywhere, at the author's whim, requiring a more flexible and forgiving analysis than would be the case with higher-level code generated by a compiler. However, if instructions and data can be distinguished, the resulting disassembly will be correct.

SHAREM utilizes Capstone to generate disassembly, using it only on regions of bytes classified as instructions. Thus, if there is a section of nine bytes classified as instructions, followed by four bytes of data, followed by six bytes of instructions, etc., Capstone will be used precisely on regions classified as instructions. If Capstone disassembled the entire shellcode, from start to end, the disassembly would quickly become inaccurate, as all bytes would be interpreted as instructions, until reaching a byte sequence that did not correspond to an instruction.

As SHAREM deals exclusively with shellcode, its approach to disassembly is more empirical and based on experimentation, rather than trying to mimic existing disassembly techniques used by *IDA Pro* or *Ghidra* analysis phases, as they clearly do not work well with shellcode. We have embraced the iterative design science research model [8]. Numerous modern *Windows* shellcode samples were scrutinized closely and used as guides, allowing for flaws in disassembly to be discovered and remediated. Once disassembly was generated, it was analysed to discover inaccuracies. The root cause for the incorrectness was then determined. All future flaws stemming from the root cause could be eradicated, by addressing what caused the disassembly to be wrong. Thus, analysis phases were designed and refined repeatedly, to eliminate entire categories of flaws. The end result was analysis phases that significantly enhanced the quality of the disassembly.

SHAREM maintains metadata about each byte, labelling each as instructions or data. If data, metadata indicates its purpose, e.g. string, WinAPI pointer, unknown, etc. When a disassembly is generated, the metadata for each byte is used to generate an accurate disassembly. With each contiguous set of bytes marked as instructions, SHAREM generates the disassembly with Capstone; bytes marked as data are handled by custom functions. If each byte's metadata is accurate, then the disassembly generated will be devoid of errors.

SHAREM provides unique analysis phases to differentiate between instructions and data. The following comes from some of the analysis phases:

- **Finding repeating data bytes**. In this phase, repeating patterns of 00s and FFs are searched for. If four or more repeat, these are marked as data, used as padding. Where there are fewer, additional analysis occurs, to see if the byte pattern of FFs corresponds to a likely instruction that would make sense. If likely to be instructions, then SHAREM does nothing.
- **Checking for valid jump destinations**. One simple way to determine if a byte is incorrectly labelled as code is to generate a disassembly beginning after the last known region of data ended. Each branching instruction is evaluated to see if its destination is possible to exist. If not, then the bytes producing the branching instruction are marked as data. While imperfect, this frequently helps correct some incorrect disassembly.
- **Locating hidden calls and jumps**. Shellcode may have numerous jumps and function calls in a small space. When generating disassembly, some of these may get lost. For instance, disassembly might start at an incorrect location, a jump or call may be lost, or instructions could be misclassified as data. Thus, SHAREM searches for opcodes that produce all short jumps or calls, E8, E9, and EB. If found, each is expanded and checked to see where the branching destination would be, if it were an instruction. Next, the potential destination is checked to see if it is at the start of an instruction. If not, SHAREM ensures that the offset for that destination is the beginning of an instruction. Practical experience with many shellcode samples with existing Assembly source code has shown this is an effective way of correcting errors in disassembly.
- **Strings in disassembly**. SHAREM's Strings Discovery Engine discovers ASCII, Unicode and Push Stack strings. The workings of each differ in some subtle ways from traditional implementations, to allow for the greater flexibility necessary with shellcode. DAE classifies all ASCII and Unicode strings as data. For strings formed by *push* instructions, they are represented as instructions, and a comment with the string appears.
- **Results of DAE static analysis phases**. Some of the phases described very briefly above are repeated more than once and occur in a specific order, with more complex implementations than space allows. While our approach is imperfect, our experience with numerous shellcode samples is that the disassembly has improved significantly, compared to leading disassemblers.

5.1 Pairing emulation data with DAE

While SHAREM can produce accurate disassembly without emulation, the quality of disassembly can improve to nearly flawless when paired with emulation. If shellcode is emulated, SHAREM logs the start and size of each instruction. When generating disassembly chunks, emulation data overrides what would be produced through analysis phases, if any conflicts exist. After all, if an instruction is emulated, we can definitively classify those bytes as instructions, with start location and

size. Emulation data merged with DAE results typically means the disassembly generated comes from both approaches. SHAREM also optionally provides complete code coverage, as previously described. If code coverage runs without error, we can say with certainty that all instructions found are accurate. There still could be small amounts of data being misclassified as instructions. However, tracking memory reads and writes helps classify accessed bytes as data.

5.2 Disassembly from the emulation of encoded shellcode

Looking at encoded shellcode in a disassembler typically reveals little; one sees a decoder stub and a series of encoded bytes. To view the disassembly of the encoded shellcode, one method is to step through it, line by line, in a debugger. With SHAREM, this is not necessary, as the decoded state of the shellcode is captured and immediately disassembled.

To achieve this, intermediate stages of the shellcode are saved and merged with the encoded shellcode. After each instruction is executed, its starting location and size are recorded; the transformed byte is saved as well. After emulation, SHAREM compares the original and final form of each byte. If different, the final form is taken as the merged shellcode. Then, SHAREM compares each byte of the current merged shellcode with the form of the byte when it was last executed. If dissimilar, the byte's value when last executed is its final merged form. Thus, if a shellcode were to modify bytes after execution, to prevent recovery of the decoded shellcode, SHAREM still would be successful in obtaining the decoded form of the shellcode. For many shellcode samples that are not highly sophisticated, this results in recovering the shellcode with complete accuracy. With some advanced samples continuously changing each byte, etc., this approach represents an acceptable compromise, as while the results may not be fully accurate, at least the instructions executed are accurate at the correct offset.

```
0x97 lea ebx, [edx + 0x20b]                 8d 9a 0b 02 00 00      ......
0x9d push ebx                               53                     S
0x9e push 0x80000001                        68 01 00 00 80         h....
0xa3 call dword ptr [edx + 0x1b4]           ff 92 b4 01 00 00      ......
    ;call to RegCreateKeyA
      (HKEY_CURRENT_USER,
       \Software\Microsoft\Windows\CurrentVersion\Run\fwjlei, 0x12000261 ->
       HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run\fwjlei)
```

Figure 9: Although this shellcode was heavily encoded, SHAREM automatically displayed its decrypted form, identifying a function call to establish persistence, with RegCreateKeyA.

This approach means transformed bytes is what appears in the final disassembly. For instance, API tables, initialized with null bytes, will be populated with function pointers, and SHAREM identifies what they correspond to, e.g. a pointer to VirtualAlloc, rather than a series of null bytes. When emulation data is merged with DAE results, SHAREM's output can be a unique hybrid between disassembly and debugged instructions, as some bytes are 'updated', yet everything is presented as static disassembly.

5.3 Disassembly annotations

One of SHAREM's extraordinary features is its unrivalled shellcode disassembly annotations. SHAREM identifies and explicates numerous shellcode features and *Windows* internal structures. Most importantly, SHAREM labels all functions called with parameters. Thus, an opaque *call eax* would be identified as a specific function.

```
0x5 mov eax, dword ptr fs:[ecx + 0x30]      64 8b 41 30            d.A0
    ; load TIB
0x9 mov eax, dword ptr [eax + 0xc]          8b 40 0c               .@.
    ; load PEB_LDR_DATA LoaderData
0xc mov esi, dword ptr [eax + 0x14]         8b 70 14               .p.
    ; LIST_ENTRY InMemoryOrderModuleList
0xf lodsd eax, dword ptr [esi]              ad                     .
    ; advancing DLL flink
```

Figure 10: SHAREM automatically annotates many shellcode attributes, including specific elements of walking the PEB.

5.3.1 Disassembly annotations from SIDE

SHAREM's highly flexible Shellcode Instruction Detection Engine (SIDE) is responsible for identifying many instructions frequently associated with shellcode or malware. These include GetPC instructions, Push Rets, Heaven's Gate instructions, walking the PEB, and more. SHAREM saves unique metadata for each, adding them both to the report and disassembly. We briefly discuss each category:

- GetPC instructions, such as *fstenv* and *call x / pop y (Call Pop)*, allow for self-location in memory [9]. In SHAREM, the *pop* will be identified as GetPC. With *fstevn*¸ SHAREM identifies the initial floating point to set up GetPC, and *fstenv* is labelled GetPC.

- A *push x* followed by a *ret* (Push Ret) is frequently used as a way for a shellcode to covertly change control flow, without being too obvious. A destination is pushed onto the stack; the *ret* causes EIP to return there. SHAREM identifies the location where the *push* occurs and the *ret*.
- Heaven's Gate instructions [10], while not common in shellcode, cause 32-bit instructions being executed to transition to 64-bit code. The intent is obfuscation, confusing analysts and causing most debuggers to malfunction [11]. For Heaven's Gate to work, the 0x33 reserved selector must be specified and a far jump or far call must be made, or the 0x33 must be pushed, followed by a *retf*. Whatever the approach, SHAREM identifies the 0x33 segment selector and the offset where the Heaven's Gate technique is invoked, labelling each.
- PEB walking is ubiquitous in modern shellcode using WinAPIs. We have identified hundreds of possible variations. Additionally, we created a points system to ensure sufficient elements of PEB walking are present, ruling out false positives. SHAREM identifies relevant elements of PEB walking, such as loading TIB, loading the PEB_LDR_DATA Loader Data, which specific LIST_Entry module list is being loaded, or advancing a DLL 'flink'.

While SIDE gives insights into shellcode, the identification of not only WinAPIs and *Windows* syscalls being called, but their actual parameter values, has never been achieved for shellcode disassembly. Sophisticated shellcode doesn't use plaintext to load WinAPIs, but instead uses checksums. With SHAREM, all functions are visible.

5.3.2 API tables

Shellcode often uses API tables to store API pointers, once they have been discovered from the exports table. SHAREM analyses memory after emulation, labelling pointers to functions in the disassembly. This is more meaningful, and the data is not misclassified as instructions, causing incorrect disassembly.

```
0x1c8 CreateProcessA - API pointer          26 c4 25 14          &.%.
0x1cc DeleteFileA - API pointer       e0 07 26 14          ..&.
0x1d0 ExitProcess - API pointer       ac 2d 26 14          .-&.
0x1d4 LoadLibraryA - API pointer            73 fd 25 14          s.%.
0x1d8 Sleep - API pointer             b3 c4 25 14          ..%.
0x1dc VirtualAlloc - API pointer            0a cc 25 14          ..%.
```

Figure 11: SHAREM identifies the API table with function pointers in the disassembly.

6. OTHER SHAREM OPERATIONS

6.1 Input files

SHAREM analyses both shellcode and PE files; SHAREM emulates and analyses 32- and 64-bit shellcode Acceptable shellcode input is binary format or ASCII characters in a text file. SHAREM can operate on a single file or a directory. PE files, while not a focus of SHAREM, can be analysed to discover common shellcode or malware attributes, indicating possible shellcode embedded within; basic PE parsing is performed. PE file emulation is outside this project's scope.

6.2 Hashes

SHAREM hashes shellcodes with SHA-256, MD5 and SSDeep. Additionally, if shellcode is deobfuscated, SHAREM hashes both the encrypted and decrypted form, including the decoder stub and decoded body, allowing for significant correlations to be made outside of SHAREM.

6.3 Reporting of findings

SHAREM is both a command line tool and usable by other frameworks, such as web services, with submitted samples. SHAREM provides numerous outputs. Firstly, results are printed to the console, in a colourful, aesthetically pleasing fashion. For each sample, SHAREM creates a directory based on filename. SHAREM saves a highly detailed report, in both text and JSON format, in the Logs directory. How SHAREM operates is customizable via the console or config file, giving flexibility for what information is gathered and how it is obtained. SHAREM provides several useful outputs, such as disassembly of the shellcode, as text and JSON. Different parts of disassembly are put into JSON structures, allowing attributes to be displayed in a customizable way via third-party applications. SHAREM additionally outputs shellcode both in binary format and as ASCII text. For deobfuscated samples, SHAREM outputs their decrypted form in binary format, with decoder stub removed. Finally, SHAREM automatically creates a .c tester file, allowing the user to compile a harness program with the shellcode inserted, allowing it to be run in a debugger. A breakpoint (INT 3) is inserted immediately before the shellcode begins.

6.4 Integration with web services

SHAREM's ability to save the shellcode's attributes to JSON makes it well-suited for integration with various applications, including web services. While SHAREM provides numerous customizable options, they are settable via the config file, and one option allows for SHAREM to be run without input or output to the console. SHAREM simply generates output.

SHAREM was designed concurrently with SubParse, a tool presented at Black Hat Arsenal 2022. SHAREM and SubParse were funded by the same NSA/NCAE grant. SubParse is a customizable, modular automated malware framework to facilitate static and dynamic analysis of submitted samples, including PE, ELF, OLE, RTF and shellcode. SHAREM acts as the shellcode parser. SubParse aggregates results into a database, viewable via web application, allowing dynamic searching, and pivoting off any indexed data field, to discover correlations among samples.

SubParse uses subprocess calls to interact with SHAREM, allowing SHAREM to emulate and analyse shellcode samples. SubParse utilizes a SharemWrapper object to ingest the JSON files created by SHAREM into an *Elasticsearch* database, allowing for dynamic search capabilities and correlation, and displaying SHAREM output via web application.

SHAREM is well-suited for adoption by other web services to provide automated analysis of shellcode samples. The documentation provides further information.

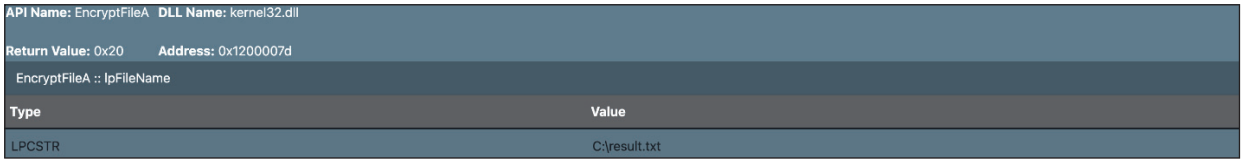

Figure 12: SubParse, a malware framework and web service, displays a WinAPI found by using SHAREM as its shellcode parser.

6.5 Shellcode classification

SHAREM determines if submitted samples are shellcode. In many cases, SHAREM can definitively classify a sample as shellcode, due to discovery of WinAPIs via emulation. In other cases, SHAREM may determine it is likely shellcode, if PEB walking features are present. Alternatively, a sample may be classified as possible shellcode or not likely shellcode.

Figure 13: SubParse uses SHAREM as its shellcode parser, determining definitively that the sample was shellcode.

7. BRUTE-FORCE DEOBFUSCATION CAPABILITIES

SHAREM provides unique brute-force deobfuscation capabilities for shellcode without using emulation. This is possible even if the decoder stub is missing. There are two requirements: first, the deobfuscation must be performed one byte at a time, with arithmetic or bitwise operations, and with one key for each. Second, PEB walking must be present, which is a requirement for most shellcode using WinAPIs. SHAREM utilizes self-modifying code to generate every permutation produced by a set of selected encoding operations. Each is checked to see if PEB walking is present; if so, the shellcode has successfully been deobfuscated.

SHAREM supports multiple encoding operations, and shellcode encoded with only one operation is deobfuscated in less than five seconds. Additionally, if a decoder stub is present, it is analysed to find possible encoding operations and keys. If found, deobfuscation occurs nearly instantaneously.

SHAREM's brute-force deobfuscation supports both parallel processing and distributed computing. Extensive testing was performed with both approaches. Even with two operations, the average time taken was under 45 seconds. For three operations, the time cost increases, and on average it takes 17 hours with parallel processing, but only 2.5 hours with distributed computing. In most cases, emulating shellcode is more effective and faster. However, in unusual cases, such as if a decoder-stub is missing, brute-force deobfuscation can be useful.

8. VALIDATION

SHAREM has analysed a vast number of shellcode samples from many sources. From these, around 200 samples were used with source code available. In other cases, CAPE sandbox with PE Sieve [12] deployed was used to acquire possible shellcode samples captured while detonating around 2,000 malware samples from MalwareBazaar and HybridAnalysis. Another set of samples came from MalwareBazaar, tagged as shellcode. A fourth dataset was created by using the PE_to_shellcode [13] tool, to convert executables into position-independent shellcode; this was done to over 200 samples of both normal PE files and malware. With some shellcode conversions, much larger in size, sometimes hundreds of APIs were logged by SHAREM. Finally, hundreds of testing shellcode samples were created, to test the efficacy of custom functions implementations being logged.

A more formal academic paper will discuss evaluation results in greater depth. In general, it can be said that SHAREM is highly effective with modern, normal shellcode samples, with respect to its various novel contributions. As bugs inevitably crop up, they are investigated, allowing for remediation.

9. OUR CONTRIBUTIONS

This research has made several contributions. We present a novel shellcode emulator, capable of logging more than 16,000 WinAPIs and nearly all user-mode *Windows* syscalls. SHAREM is uniquely able to deobfuscate shellcode, automatically presenting highly annotated disassembly of its deobfuscated form. It also directly outputs the recovered shellcode in binary format. SHAREM's complete code coverage of emulation helps ensure missing functions are identified, which otherwise would be overlooked by a tool without this feature. SHAREM has a highly accurate disassembler of shellcode. Moreover, SHAREM can merge emulation results with what is found statically, producing a disassembly that can be nearly flawless. SHAREM greatly enhances the disassembly, identifying all function calls with parameters, with many shellcode attributes also identified, such as elements of walking the PEB. Finally, SHAREM provides timeless debugging capabilities, allowing users to examine what is being done step by step, without needing to manually debug shellcode.

10. FINAL REMARKS

Our goal is for SHAREM to support all shellcode, whether 32- or 64-bit, and using WinAPIs or *Windows* syscalls. Thus, we support all functions likely to be security relevant; additional DLLs and WinAPIs could be added as needed. Shellcode closely impacts both malware and exploitation, and SHAREM brings unparalleled capabilities to malware analysts, allowing more to be done with far greater ease.

ACKNOWLEDGEMENTS

This research and all the co-authors have been supported by NSA Grant H98230-20-1-0326.

REFERENCES

[1] Mds. Research. Bypassing User-Mode Hooks and Direct Invocation of System Calls for Red Teams. MDSec, 2020. https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/.

[2] Borders, K., Prakash, A.; Zielinski, M. Spector: Automatically analyzing shell code. Proc. - Annu. Comput. Secur. Appl. Conf. ACSAC, pp. 501–514, 2007.

[3] Fratantonio, Y.; Kruegel, C.; Vigna, G. Shellzer: a tool for the dynamic analysis of malicious shellcode. International workshop on recent advances in intrusion detection, 2011, pp. 61–80.

[4] Zimmer, D. Scdbg Shellcode Analysis. 2011. http://sandsprite.com/CodeStuff/scdbg_manual/MANUAL_EN.html.

[5] FireEye. Speakeasy. https://github.com/fireeye/speakeasy.

[6] Jurczyk, M. Windows X86-64 System Call Table (XP/2003/Vista/2008/7/2012/8/10). https://j00ru.vexillium.org/syscalls/nt/64/.

[7] Nowak, T. The Undocumented Functions Microsoft Windows NT/2000/XP/Win7. NTAPI Undocumented Functions.

[8] Hevner, A. R.; March, S. T.; Park, J.; Ram, S. Design science in information systems research. MIS Q., pp. 75–105, 2004.

[9] Anley, C.; Heasman, J.; Lindner, F.; Richarte, G. The shellcoder's handbook: discovering and exploiting security holes. John Wiley & Sons, 2011.

[10] Eckels, S. WOW64!Hooks: WOW64 Subsystem Internals and Hooking Techniques. Mandiant, 2020. https://www.mandiant.com/resources/wow64-subsystem-internals-and-hooking-techniques.

[11] Ionescu, A. Closing Heaven's Gate. 2015. https://www.alex-ionescu.com/?p=300.

[12] Hasherezade. PE-Sieve. GitHub. 2018. https://github.com/hasherezade/pe-sieve.

[13] Hasherezade. PE to Shellcode. GitHub. 2021. https://github.com/hasherezade/pe_to_shellcode.