Windows Defender Under The Microscope:

A Reverse Engineer’s Perspective

Alexei Bulazel

@0xAlexei

Virus Bulletin 2018

About Me

- AV industry outsider working on AV RE for a long time
- Security researcher at ForAllSecure
- RPI / RPISEC alumnus
    - Co-taught the famous RPISEC "Modern Binary Exploitation" class (`https://github.com/rpisec/mbe`)
- First time at Virus Bulletin

This is my personal research, any views and opinions expressed are my own, not those of any employer

@0xAlexei

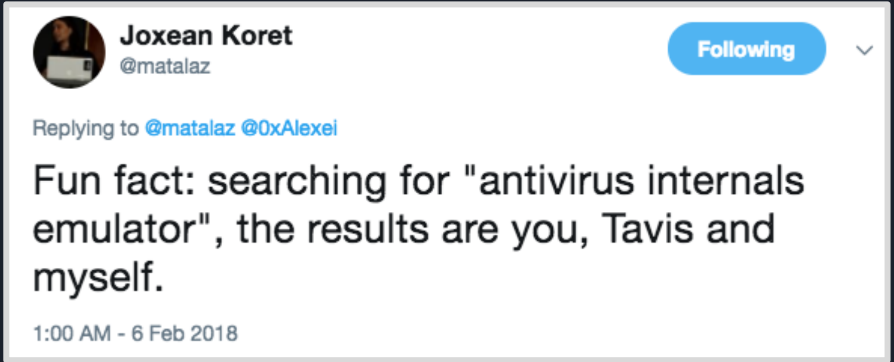
RPISEC

Outline

1. Introduction
2. Tooling & Process
3. Discussion
4. Conclusion

This Talk

- Analysis of my custom tools and process after 9+ months of REing Windows Defender
    - Not going to reiterate AV knowledge that industry already knows - see released slides



Joxean Koret
@matalaz

Replying to @matalaz @0xAlexei

Fun fact: searching for "antivirus internals emulator", the results are you, Tavis and myself.

1:00 AM - 6 Feb 2018

- Few researchers REing AVs, fewer looking at emulators
- No disrespect to Microsoft or the AV industry - Defender is a fascinating subject of study and a beautifully architected piece of software

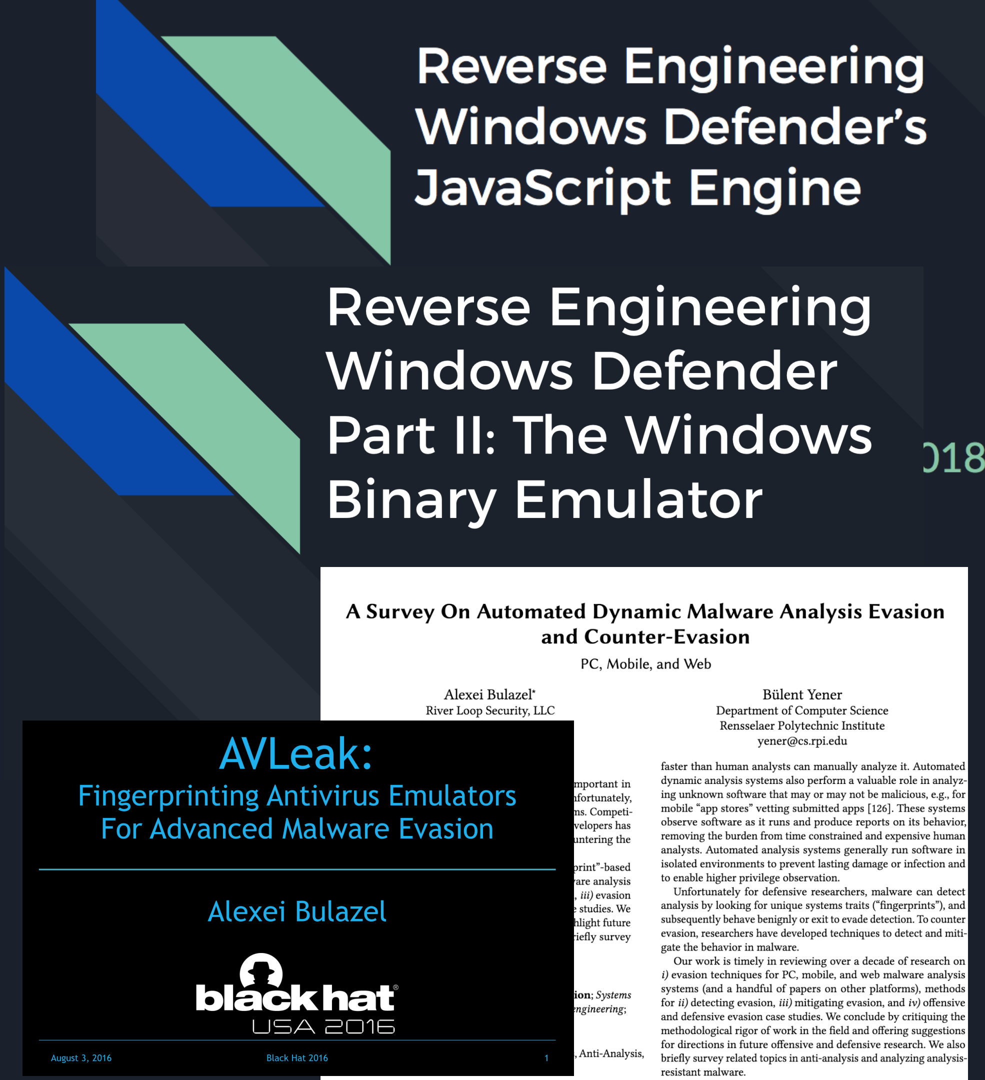
My Published Research

Windows Defender RE

- JS Engine @ REcon Brussels
- Windows Emulator @ REcon Montreal, Black Hat, DEFCON

"AVLeak" - AV emulator fingerprinting and evasion @ Black Hat & WOOT'16

"A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion" @ ROOTS'17



Reverse Engineering Windows Defender's JavaScript Engine

Reverse Engineering Windows Defender Part II: The Windows Binary Emulator

)18

A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion

PC, Mobile, and Web

Alexei Bulazel*
River Loop Security, LLC

Bülent Yener
Department of Computer Science
Rensselaer Polytechnic Institute
yener@cs.rpi.edu

mportant in
nfortunately,
ns. Competi-
velopers has
untering the

print"-based
are analysis
, *iii)* evasion
studies. We
hlight future
iefly survey

ion; *Systems*
ngineering;

, Anti-Analysis,

faster than human analysts can manually analyze it. Automated dynamic analysis systems also perform a valuable role in analyzing unknown software that may or may not be malicious, e.g., for mobile "app stores" vetting submitted apps [126]. These systems observe software as it runs and produce reports on its behavior, removing the burden from time constrained and expensive human analysts. Automated analysis systems generally run software in isolated environments to prevent lasting damage or infection and to enable higher privilege observation.

Unfortunately for defensive researchers, malware can detect analysis by looking for unique systems traits ("fingerprints"), and subsequently behave benignly or exit to evade detection. To counter evasion, researchers have developed techniques to detect and mitigate the behavior in malware.

Our work is timely in reviewing over a decade of research on *i)* evasion techniques for PC, mobile, and web malware analysis systems (and a handful of papers on other platforms), methods for *ii)* detecting evasion, *iii)* mitigating evasion, and *iv)* offensive and defensive evasion case studies. We conclude by critiquing the methodological rigor of work in the field and offering suggestions for directions in future offensive and defensive research. We also briefly survey related topics in anti-analysis and analyzing analysis-resistant malware.

AVLeak:

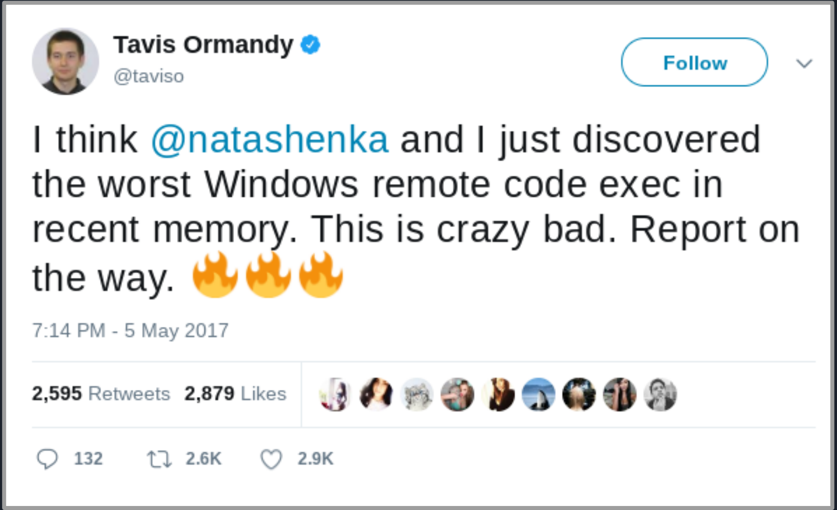
Fingerprinting Antivirus Emulators For Advanced Malware Evasion

Alexei Bulazel

black hat USA 2016

August 3, 2016 Black Hat 2016 1

Motivation



Tavis Ormandy @taviso

I think @natashenka and I just discovered the worst Windows remote code exec in recent memory. This is crazy bad. Report on the way. 🔥🔥🔥

7:14 PM - 5 May 2017

2,595 Retweets 2,879 Likes

- Tavis and co. at P0 dropped some awesome Defender JS engine bugs
- I had analyzed AVs before, but never Windows Defender… interest in JS engines
- So I reverse engineered Defender's JS engine for ~4 months
- I then spent another ~5 months reverse engineering the Windows binary emulator
- This was a *personal* research project - all in my free time, not for any company

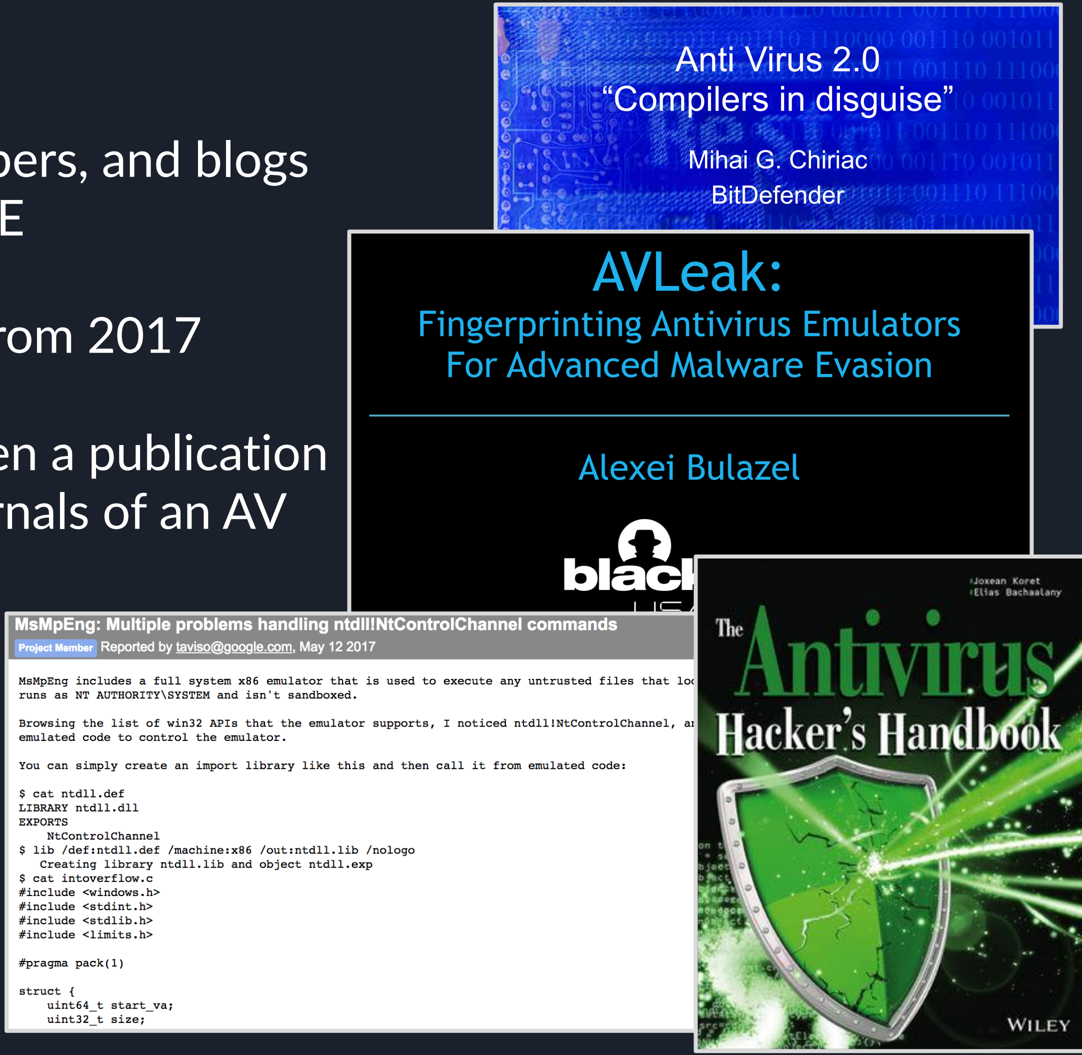
Real Motivation

Spend hundreds of hours doing unpaid research, so I can fly thousands of miles in coach class to present Powerpoints in hotels around the world

Prior Art

- Lots of conference talks, whitepapers, and blogs on antivirus *evasion*, but few on RE
- Tavis Ormandy's Defender bugs from 2017
- As far as I know, there's never been a publication about *reverse engineering* the internals of an AV emulator*

*There are plenty on black box AV evasion though. AV industry companies have occasionally presented on the design of their emulators at conferences such as Virus Bulletin.



MsMpEng: Multiple problems handling ntdll!NtControlChannel commands

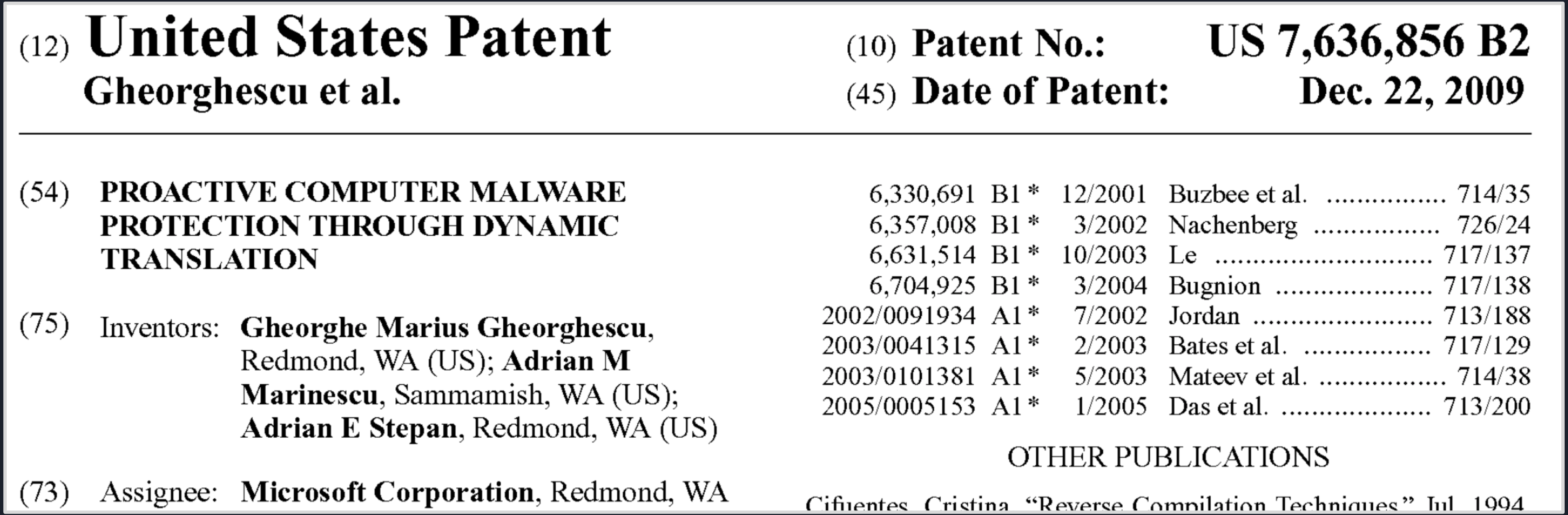
Project Member Reported by taviso@google.com, May 12 2017

```
MsMpEng includes a full system x86 emulator that is used to execute any untrusted files that loc
runs as NT AUTHORITY\SYSTEM and isn't sandboxed.

Browsing the list of win32 APIs that the emulator supports, I noticed ntdll!NtControlChannel, ar
emulated code to control the emulator.

You can simply create an import library like this and then call it from emulated code:

$ cat ntdll.def
LIBRARY ntdll.dll
EXPORTS
    NtControlChannel
$ lib /def:ntdll.def /machine:x86 /out:ntdll.lib /nologo
   Creating library ntdll.lib and object ntdll.exp
$ cat intoverflow.c
#include <windows.h>
#include <stdint.h>
#include <stdlib.h>
#include <limits.h>

#pragma pack(1)

struct {
    uint64_t start_va;
    uint32_t size;
```

Outline

1. Introduction
2. Tooling & Process
    a. Introduction
    b. JS Engine
    c. Emulator
3. Discussion
4. Conclusion

Reconnaissance - Patent Search



(12) **United States Patent**
Gheorghescu et al.

(10) **Patent No.:** **US 7,636,856 B2**
(45) **Date of Patent:** **Dec. 22, 2009**

(54) **PROACTIVE COMPUTER MALWARE PROTECTION THROUGH DYNAMIC TRANSLATION**

(75) Inventors: **Gheorghe Marius Gheorghescu**, Redmond, WA (US); **Adrian M Marinescu**, Sammamish, WA (US); **Adrian E Stepan**, Redmond, WA (US)

(73) Assignee: **Microsoft Corporation**, Redmond, WA

6,330,691	B1 *	12/2001	Buzbee et al.	714/35
6,357,008	B1 *	3/2002	Nachenberg	726/24
6,631,514	B1 *	10/2003	Le	717/137
6,704,925	B1 *	3/2004	Bugnion	717/138
2002/0091934	A1 *	7/2002	Jordan	713/188
2003/0041315	A1 *	2/2003	Bates et al.	717/129
2003/0101381	A1 *	5/2003	Mateev et al.	714/38
2005/0005153	A1 *	1/2005	Das et al.	713/200

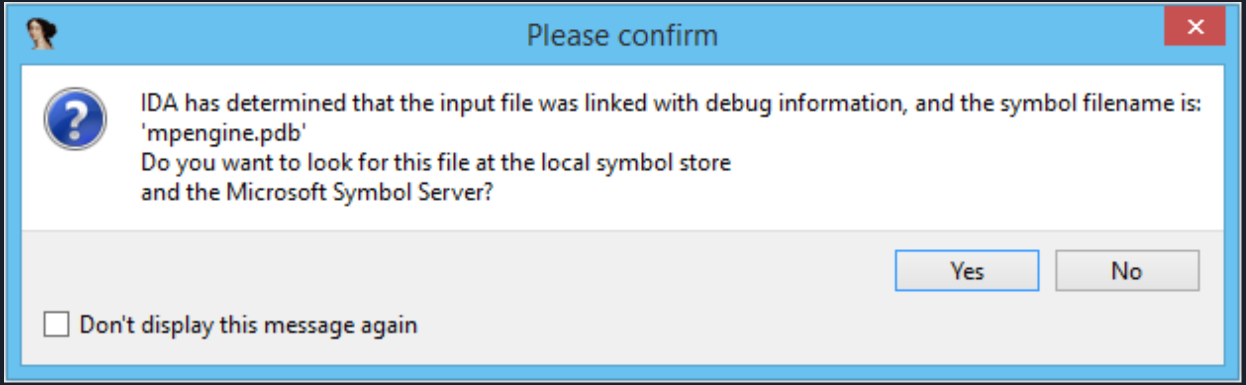
OTHER PUBLICATIONS

Cifuentes, Cristina, “Reverse Compilation Techniques,” Jul. 1994

“The present invention includes a system and method for translating potential malware devices into safe program code. The potential malware is translated from any one of a number of different types of source languages, including, but not limited to, native CPU program code, platform independent .NET byte code, scripting program code, and the like. Then the translated program code is compiled into program code that may be understood and executed by the native CPU…”

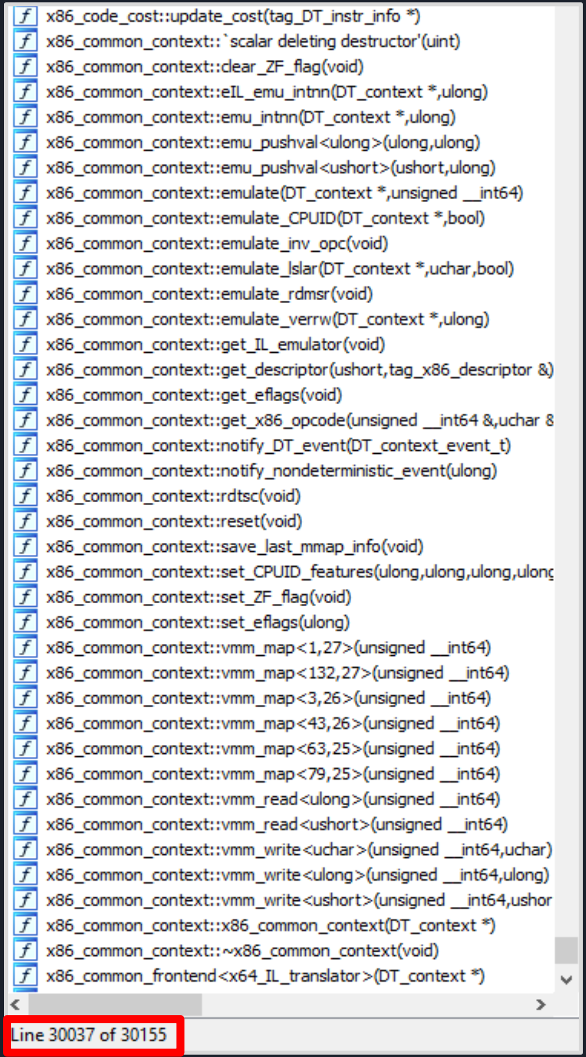
Static Analysis

- ~12 MB DLL
- ~30,000 functions
- IDA Pro
  - Patch analysis with BinDiff
- Microsoft publishes PDBs



IDA has determined that the input file was linked with debug information, and the symbol filename is:
'mpengine.pdb'
Do you want to look for this file at the local symbol store
and the Microsoft Symbol Server?

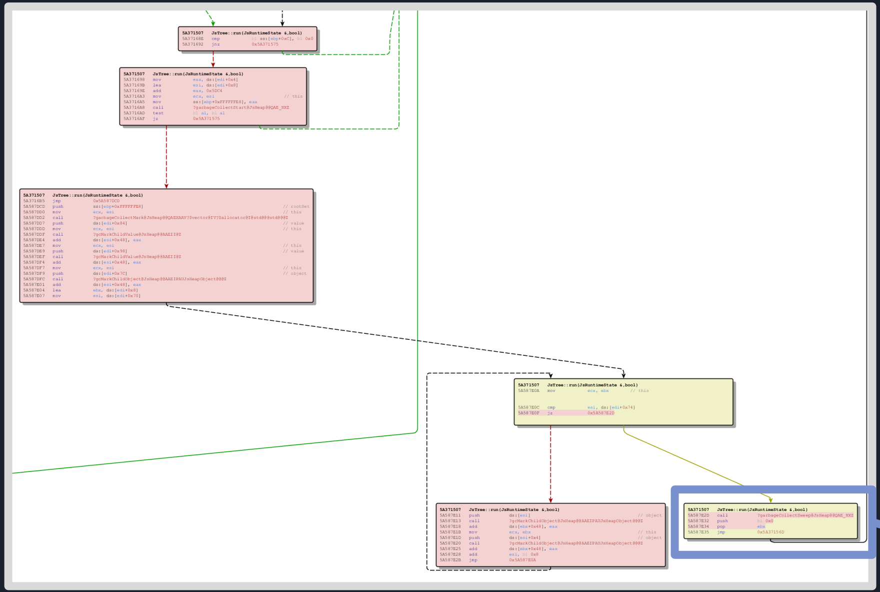
□ Don't display this message again



```
x86_code_cost::update_cost(tag_DT_instr_info *)
x86_common_context::`scalar deleting destructor'(uint)
x86_common_context::clear_ZF_flag(void)
x86_common_context::eIL_emu_intnn(DT_context *,ulong)
x86_common_context::emu_intnn(DT_context *,ulong)
x86_common_context::emu_pushval<ulong>(ulong,ulong)
x86_common_context::emu_pushval<ushort>(ushort,ulong)
x86_common_context::emulate(DT_context *,unsigned __int64)
x86_common_context::emulate_CPUID(DT_context *,bool)
x86_common_context::emulate_inv_opc(void)
x86_common_context::emulate_lslar(DT_context *,uchar,bool)
x86_common_context::emulate_rdmsr(void)
x86_common_context::emulate_verrw(DT_context *,ulong)
x86_common_context::get_IL_emulator(void)
x86_common_context::get_descriptor(ushort,tag_x86_descriptor &)
x86_common_context::get_eflags(void)
x86_common_context::get_x86_opcode(unsigned __int64 &,uchar &
x86_common_context::notify_DT_event(DT_context_event_t)
x86_common_context::notify_nondeterministic_event(ulong)
x86_common_context::rdtsc(void)
x86_common_context::reset(void)
x86_common_context::save_last_mmap_info(void)
x86_common_context::set_CPUID_features(ulong,ulong,ulong,ulong
x86_common_context::set_ZF_flag(void)
x86_common_context::set_eflags(ulong)
x86_common_context::vmm_map<1,27>(unsigned __int64)
x86_common_context::vmm_map<132,27>(unsigned __int64)
x86_common_context::vmm_map<3,26>(unsigned __int64)
x86_common_context::vmm_map<43,26>(unsigned __int64)
x86_common_context::vmm_map<63,25>(unsigned __int64)
x86_common_context::vmm_map<79,25>(unsigned __int64)
x86_common_context::vmm_read<ulong>(unsigned __int64)
x86_common_context::vmm_read<ushort>(unsigned __int64)
x86_common_context::vmm_write<uchar>(unsigned __int64,uchar)
x86_common_context::vmm_write<ulong>(unsigned __int64,ulong)
x86_common_context::vmm_write<ushort>(unsigned __int64,ushor
x86_common_context::x86_common_context(DT_context *)
x86_common_context::~x86_common_context(void)
x86_common_frontend<x64_IL_translator>(DT_context *)
```

Line 30037 of 30155

BinDiffing



```
5A371507   JsTree::run(JsRuntimeState &,bool)
5A37168E   cmp          bl ss:[ebp+0xC], bl 0x0
5A371692   jnz          0x5A371575
```

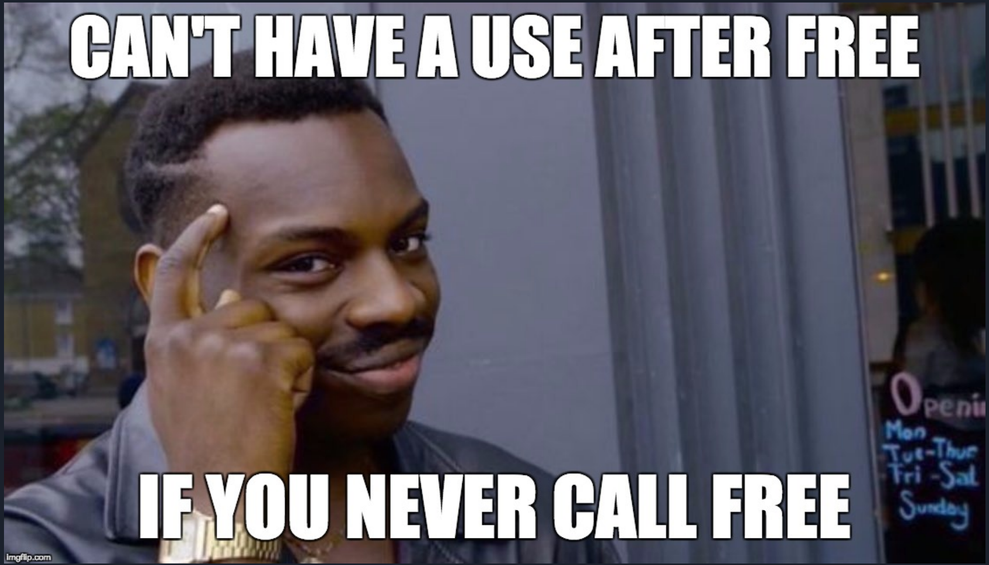
```
5A371507   JsTree::run(JsRuntimeState &,bool)
5A371698   mov          eax, ds:[edi+0x4]
5A37169B   lea          esi, ds:[edi+0x8]
5A37169E   add          eax, 0x5DC4
5A3716A3   mov          ecx, esi                                  // this
5A3716A5   mov          ss:[ebp+0xFFFFFFE8], eax
5A3716A8   call         ?garbageCollectStart@JsHeap@@QAE_NXZ
5A3716AD   test         bl al, bl al
5A3716AF   jz           0x5A371575
```

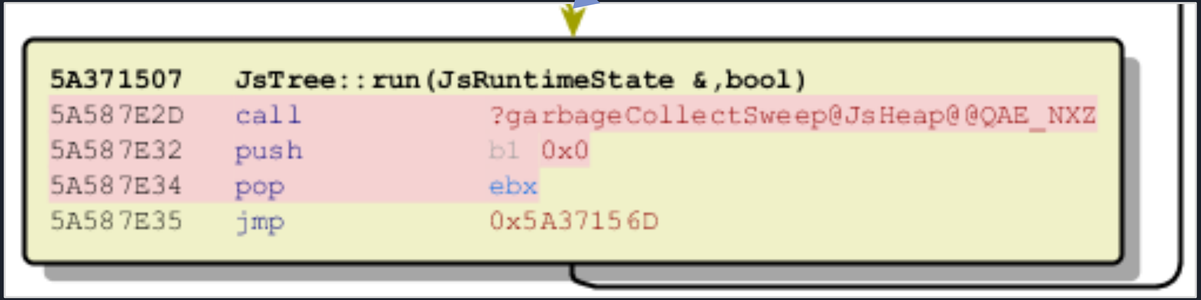
```
5A371507   JsTree::run(JsRuntimeState &,bool)
5A3716B5   jmp          0x5A587DCD
5A587DCD   push         ss:[ebp+0xFFFFFFE8]                       // rootSet
5A587DD0   mov          ecx, esi                                  // this
5A587DD2   call         ?garbageCollectMark@JsHeap@@QAEXAAV?$vector@IV?$allocator@I@std@@@std@@@Z
5A587DD7   push         ds:[edi+0x84]                             // value
5A587DDD   mov          ecx, esi                                  // this
5A587DDF   call         ?gcMarkChildValue@JsHeap@@AAEII@Z
5A587DE4   add          ds:[esi+0x48], eax
5A587DE7   mov          ecx, esi                                  // this
5A587DE9   push         ds:[edi+0x98]                             // value
5A587DEF   call         ?gcMarkChildValue@JsHeap@@AAEII@Z
5A587DF4   add          ds:[esi+0x48], eax
5A587DF7   mov          ecx, esi                                  // this
5A587DF9   push         ds:[edi+0x7C]                             // object
5A587DFC   call         ?gcMarkChildObject@JsHeap@@AAEIPAUJsHeapObject@@@Z
5A587E01   add          ds:[esi+0x48], eax
5A587E04   lea          ebx, ds:[edi+0x8]
5A587E07   mov          esi, ds:[edi+0x70]
```

```
5A371507   JsTree::run(JsRuntimeState &,bool)
5A587E0A   mov          ecx, ebx        // this


5A587E0C   cmp          esi, ds:[edi+0x74]
5A587E0F   jz           0x5A587E2D
```

```
5A371507   JsTree::run(JsRuntimeState &,bool)
5A587E11   push         ds:[esi]                                  // object
5A587E13   call         ?gcMarkChildObject@JsHeap@@AAEIPAUJsHeapObject@@@Z
5A587E18   add          ds:[ebx+0x48], eax
5A587E1B   mov          ecx, ebx                                  // this
5A587E1D   push         ds:[esi+0x4]                              // object
5A587E20   call         ?gcMarkChildObject@JsHeap@@AAEIPAUJsHeapObject@@@Z
5A587E25   add          ds:[ebx+0x48], eax
5A587E28   add          esi, bl 0x8
5A587E2B   jmp          0x5A587E0A
```

```
5A371507   JsTree::run(JsRuntimeState &,bool)
5A587E2D   call         ?garbageCollectSweep@JsHeap@@QAE_NXZ
5A587E32   push         bl 0x0
5A587E34   pop          ebx
5A587E35   jmp          0x5A37156D
```



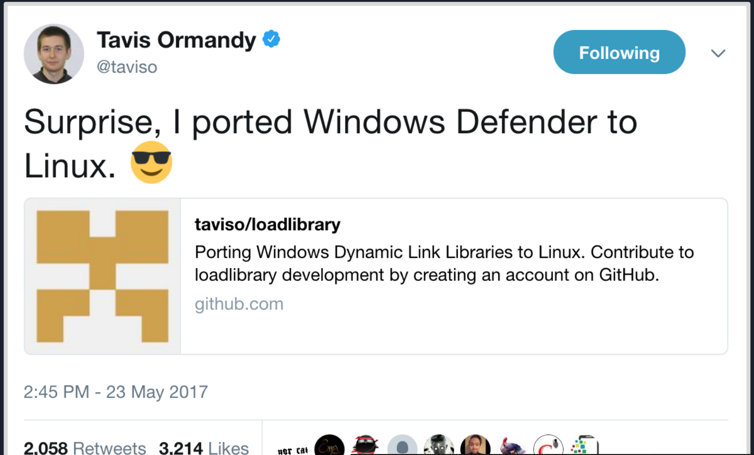


```
5A371507     JsTree::run(JsRuntimeState &,bool)
5A587E2D     call           ?garbageCollectSweep@JsHeap@@QAE_NXZ
5A587E32     push           bl 0x0
5A587E34     pop            ebx
5A587E35     jmp            0x5A37156D
```

Dynamic Analysis & Loader

AV-Specific Challenges:

- Protected Process
  - Cannot debug, even as local admin
- Introspection
- Scanning on demand
- Code reachability may be configuration / heuristics dependent



Tavis Ormandy
@taviso

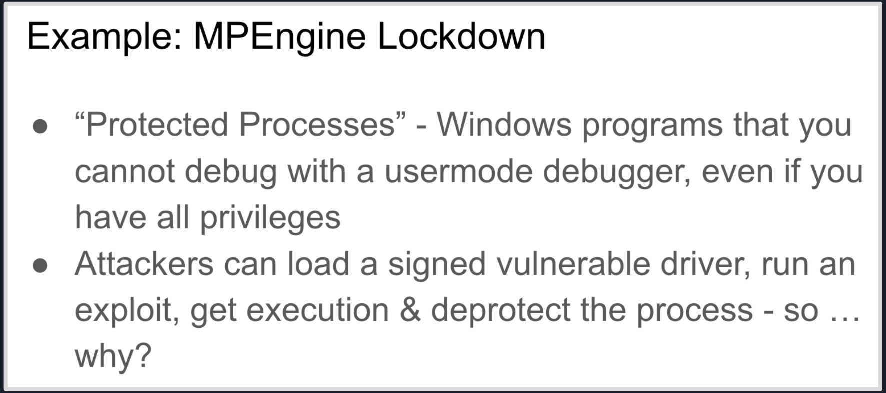
Surprise, I ported Windows Defender to Linux. 😎

taviso/loadlibrary
Porting Windows Dynamic Link Libraries to Linux. Contribute to loadlibrary development by creating an account on GitHub.
github.com

2:45 PM - 23 May 2017

2,058 Retweets 3,214 Likes

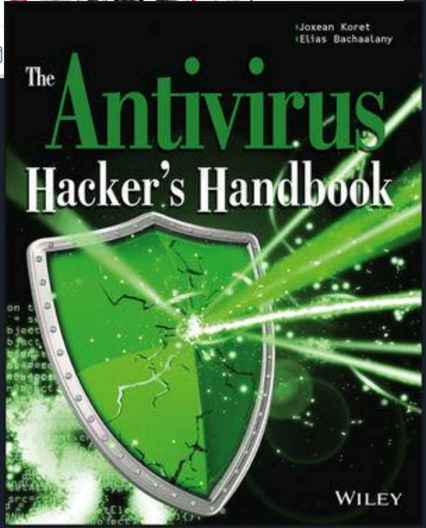
139 2.1K 3.2K



Example: MPEngine Lockdown

- "Protected Processes" - Windows programs that you cannot debug with a usermode debugger, even if you have all privileges
- Attackers can load a signed vulnerable driver, run an exploit, get execution & deprotect the process - so … why?

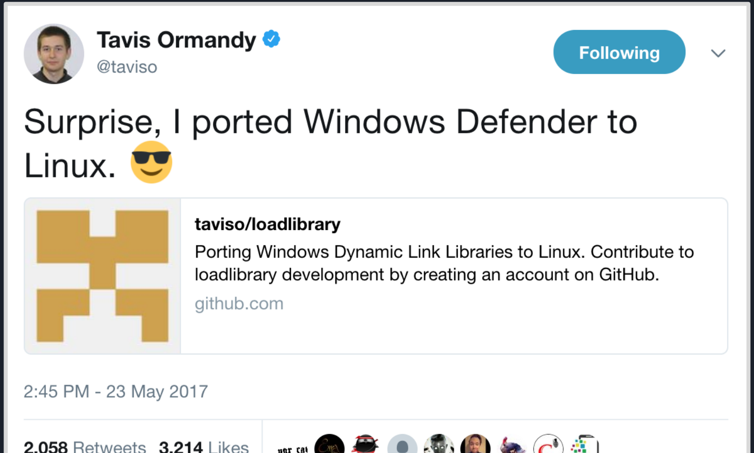
"Repeated vs. single-round games in security"
Halvar Flake, BSides Zurich Keynote



Dynamic Analysis & Loader

AV-Specific Challenges:

- Protected Process
  - Cannot debug, even as local admin
- Introspection
- Scanning on demand
- Code reachability may be configuration / heuristics dependent



Tavis Ormandy @taviso

Following

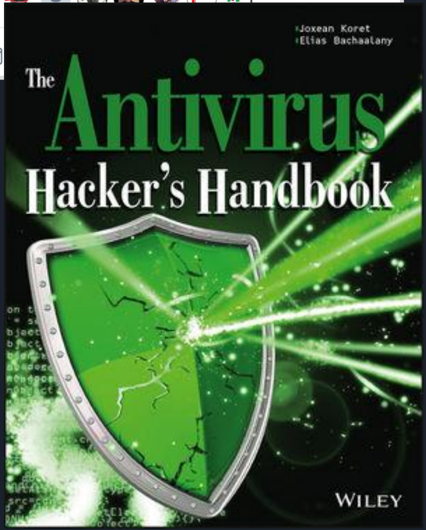
Surprise, I ported Windows Defender to Linux. 😎

taviso/loadlibrary
Porting Windows Dynamic Link Libraries to Linux. Contribute to loadlibrary development by creating an account on GitHub.
github.com

2:45 PM - 23 May 2017

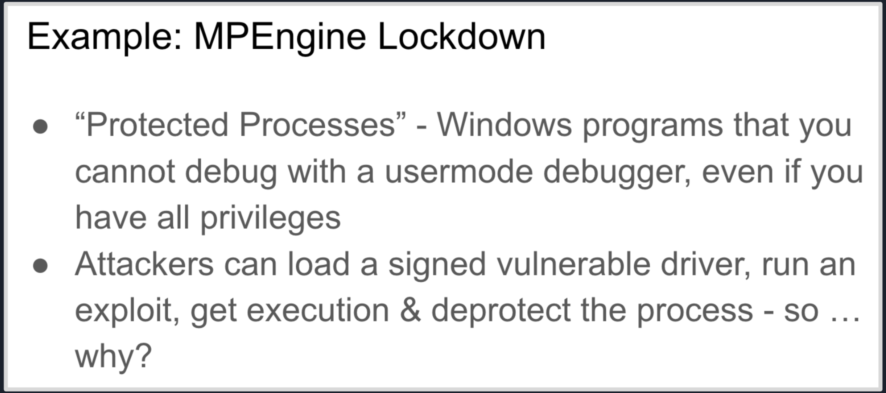
2,058 Retweets 3,214 Likes

139 2.1K 3.2K



Solution:

Custom loaders for AV binaries



Example: MPEngine Lockdown

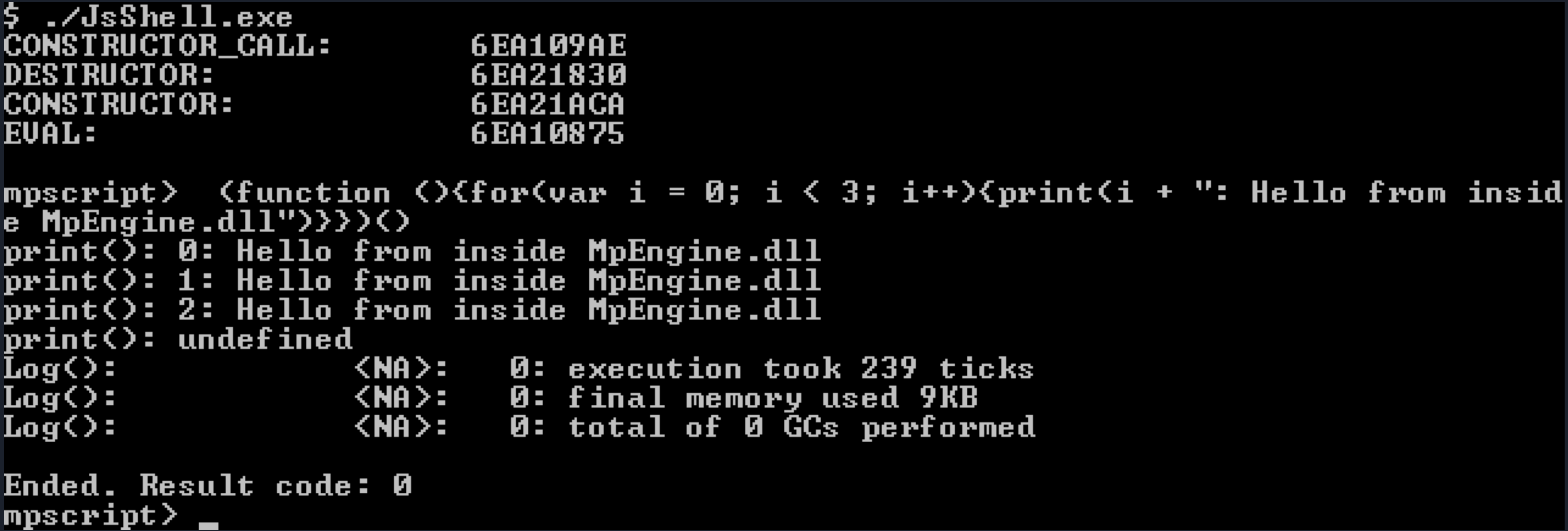
- “Protected Processes” - Windows programs that you cannot debug with a usermode debugger, even if you have all privileges
- Attackers can load a signed vulnerable driver, run an exploit, get execution & deprotect the process - so … why?

“Repeated vs. single-round games in security”
Halvar Flake, BSides Zurich Keynote

Outline

1. Introduction
2. Tooling & Process
    a. Introduction
    b. JS Engine
    c. Emulator
3. Discussion
4. Conclusion

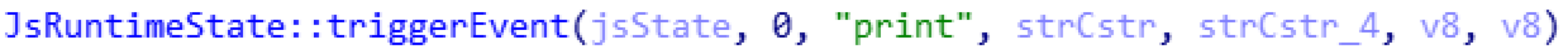
JS REPL Shell



```
$ ./JsShell.exe
CONSTRUCTOR_CALL:        6EA109AE
DESTRUCTOR:              6EA21830
CONSTRUCTOR:             6EA21ACA
EVAL:                    6EA10875

mpscript>  (function (){for(var i = 0; i < 3; i++){print(i + ": Hello from insid
e MpEngine.dll")}})()
print(): 0: Hello from inside MpEngine.dll
print(): 1: Hello from inside MpEngine.dll
print(): 2: Hello from inside MpEngine.dll
print(): undefined
Log():           <NA>:   0: execution took 239 ticks
Log():           <NA>:   0: final memory used 9KB
Log():           <NA>:   0: total of 0 GCs performed

Ended. Result code: 0
mpscript> _
```

Based off a shell released on Twitter by `@TheWack0lian`, developed with Rolf Rolles

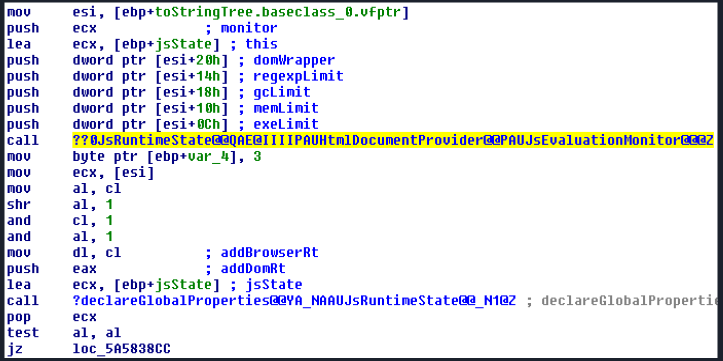
JS Loader and Shell



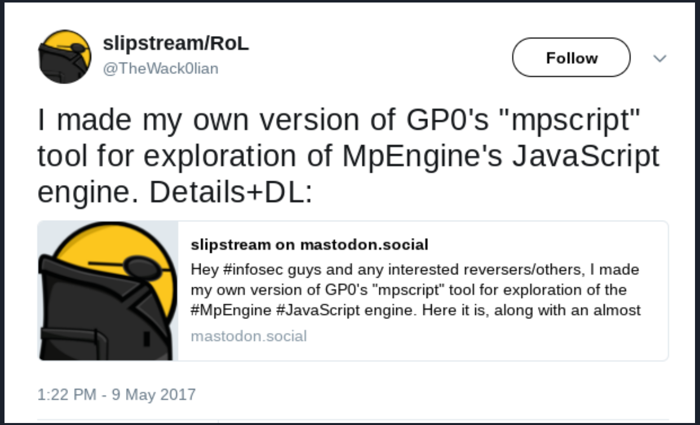
```
JsRuntimeState::triggerEvent(jsState, 0, "print", strCstr, strCstr_4, v8, v8)
```

- Use `LoadLibrary` on Windows
  - WinDbg works natively
- Patch constructor for `JsRuntimeState::JsRuntimeState()`
  - Provide a VTable implementing analysis callbacks
  - Print to `stdout` on “`print`” events
  - Log other events
- Directly call to start scan:

```
JavaScriptInterpreter::eval(
    const char *input,
    unsigned int inputSize,
    JavaScriptInterpreter::Params *params)
```



```
mov     esi, [ebp+toStringTree.baseclass_0.vfptr]
push    ecx             ; monitor
lea     ecx, [ebp+jsState] ; this
push    dword ptr [esi+20h] ; domWrapper
push    dword ptr [esi+14h] ; regexpLimit
push    dword ptr [esi+18h] ; gcLimit
push    dword ptr [esi+10h] ; memLimit
push    dword ptr [esi+0Ch] ; exeLimit
call    ??0JsRuntimeState@@QAE@IIIIPAUHtmlDocumentProvider@@PAUJsEvaluationMonitor@@@Z
mov     byte ptr [ebp+var_4], 3
mov     ecx, [esi]
mov     al, cl
shr     al, 1
and     cl, 1
and     al, 1
mov     dl, cl          ; addBrowserRt
push    eax             ; addDomRt
lea     ecx, [ebp+jsState] ; jsState
call    ?declareGlobalProperties@@YA_NAAUJsRuntimeState@@_N1@Z ; declareGlobalPropertie
pop     ecx
test    al, al
jz      loc_5A5838CC
```



slipstream/RoL
@TheWack0lian

I made my own version of GP0's "mpscript" tool for exploration of MpEngine's JavaScript engine. Details+DL:

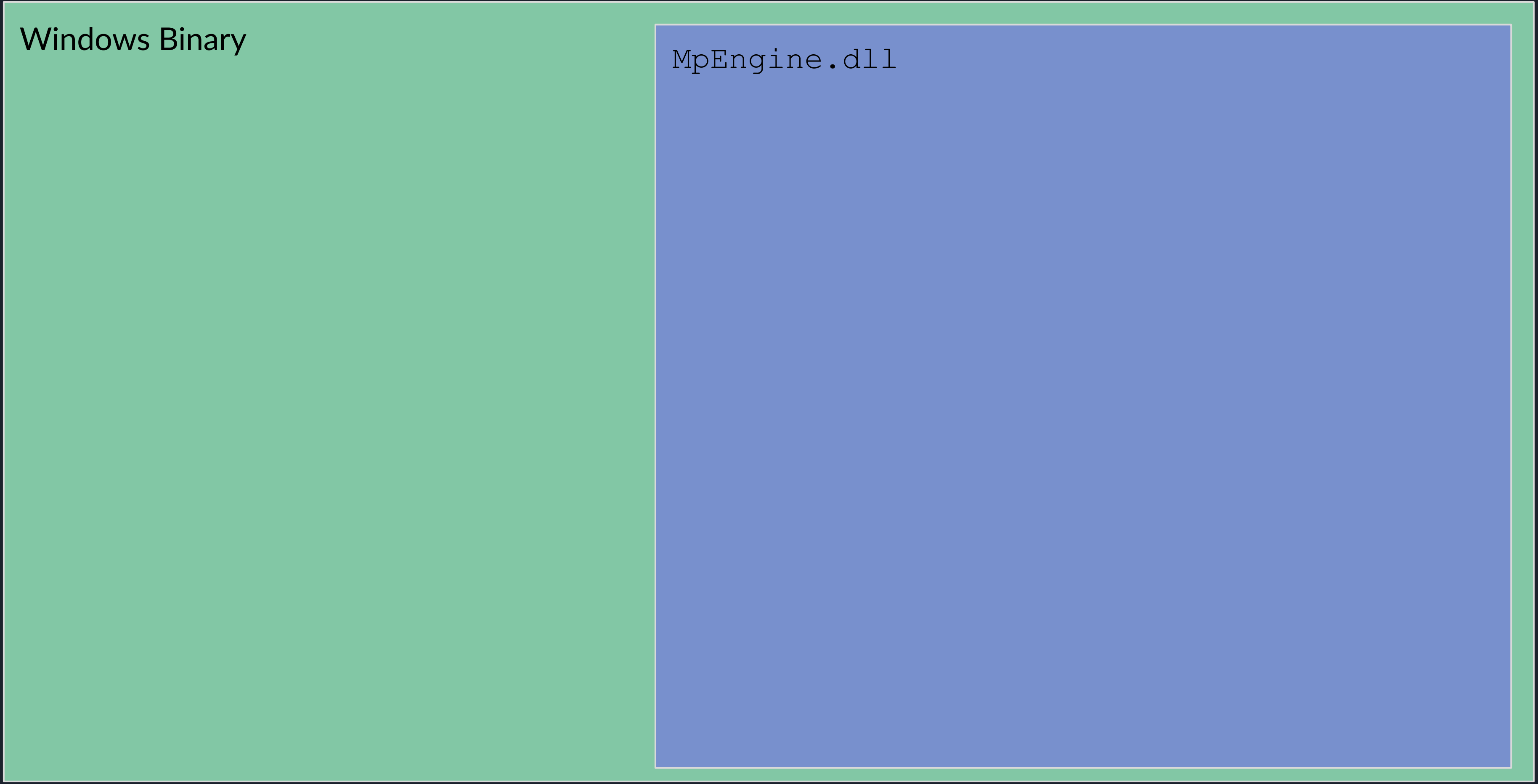
slipstream on mastodon.social
Hey #infosec guys and any interested reversers/others, I made my own version of GP0's "mpscript" tool for exploration of the #MpEngine #JavaScript engine. Here it is, along with an almost
mastodon.social

1:22 PM - 9 May 2017

JS Loader and Shell

Windows Binary

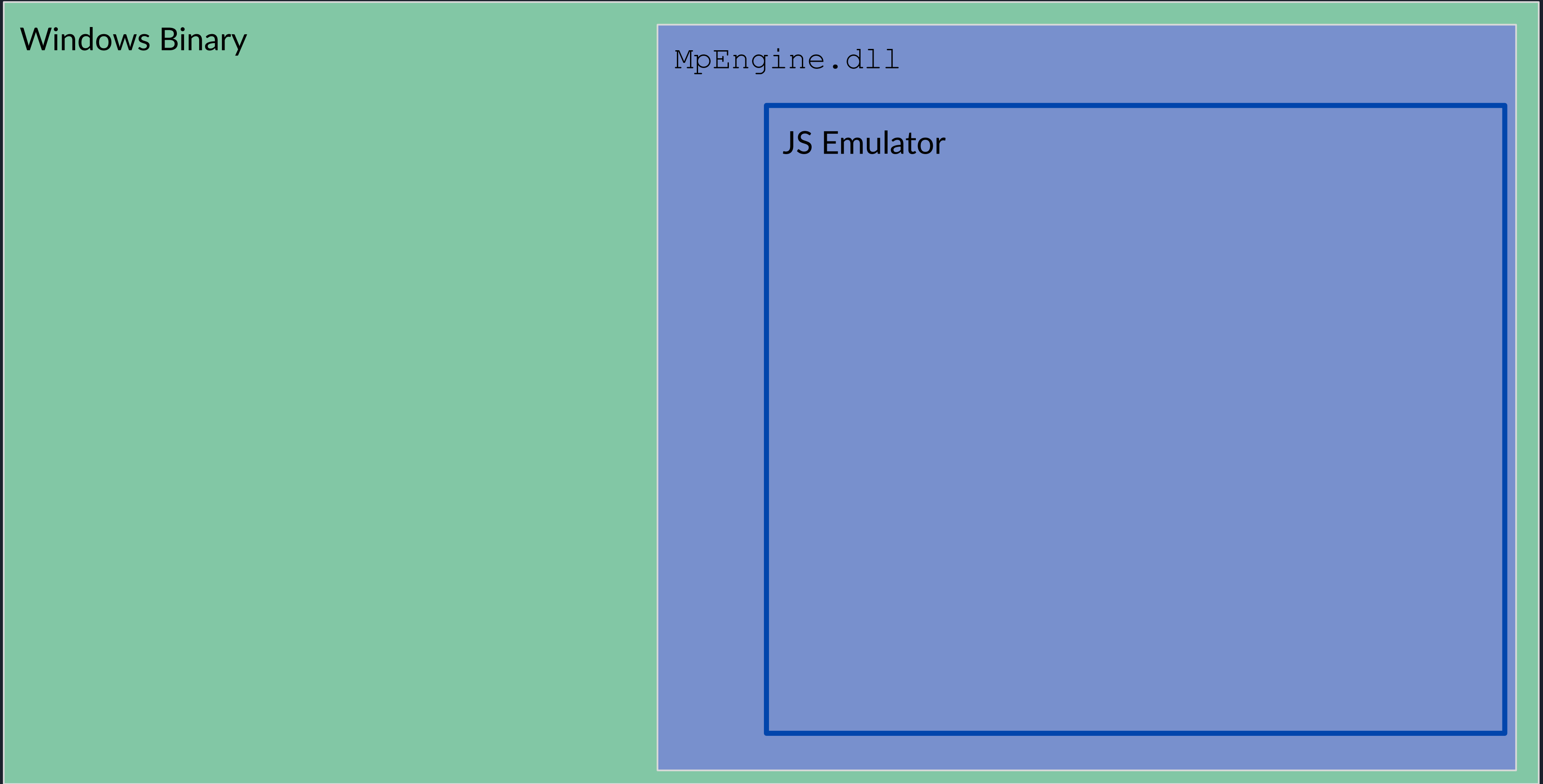
JS Loader and Shell



Windows Binary

`MpEngine.dll`

JS Loader and Shell

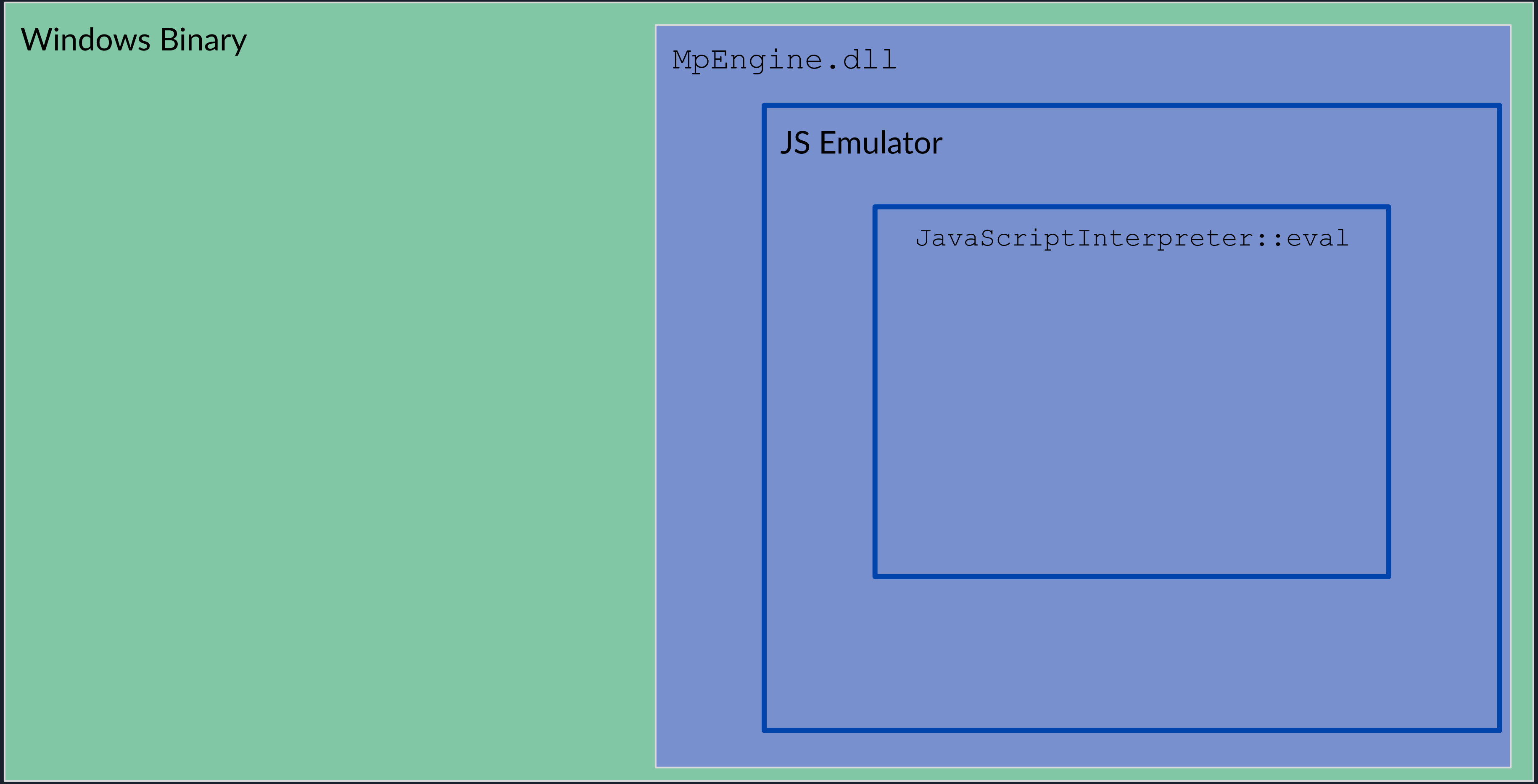


Windows Binary

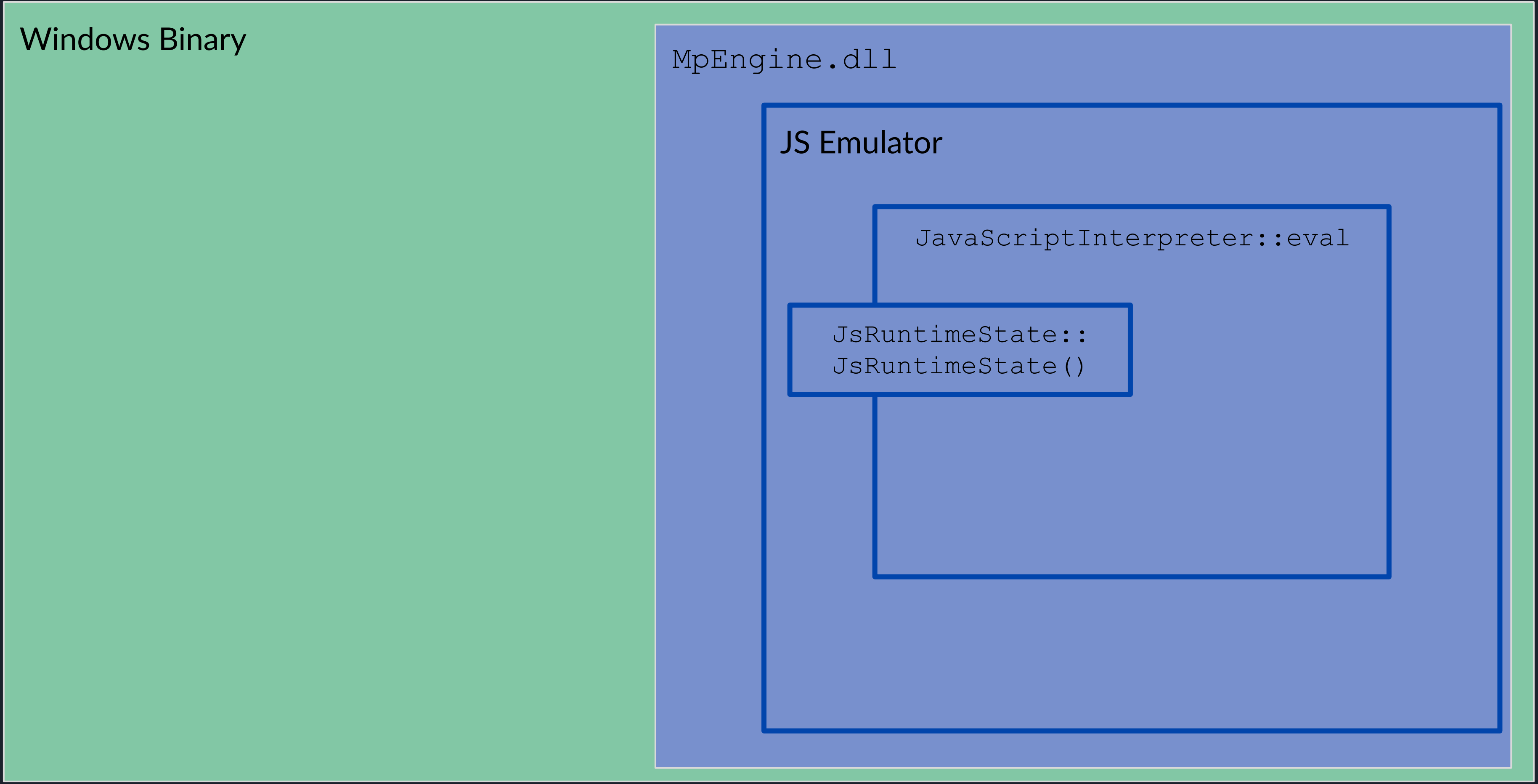
MpEngine.dll

JS Emulator

JS Loader and Shell



JS Loader and Shell



Windows Binary

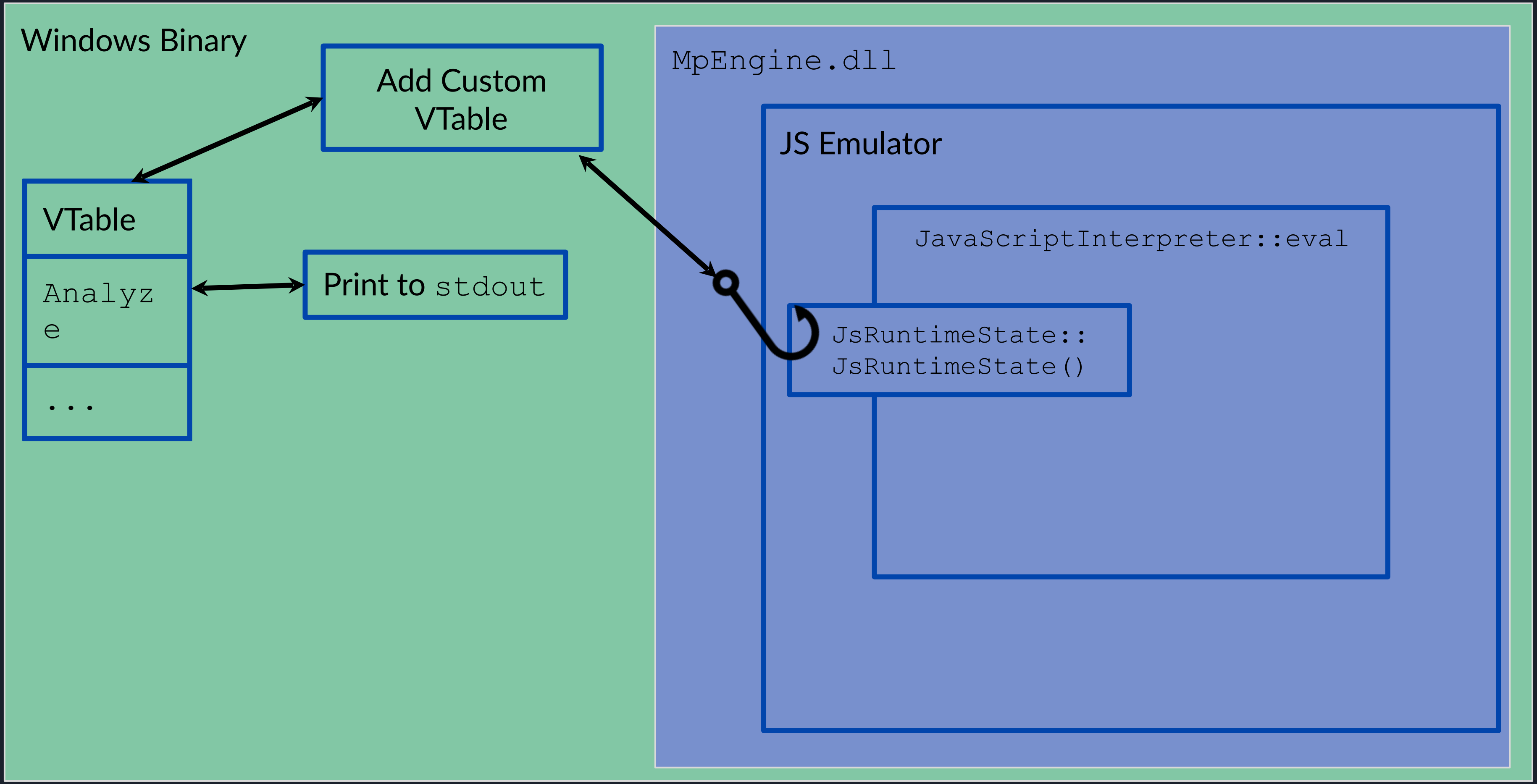
MpEngine.dll

JS Emulator

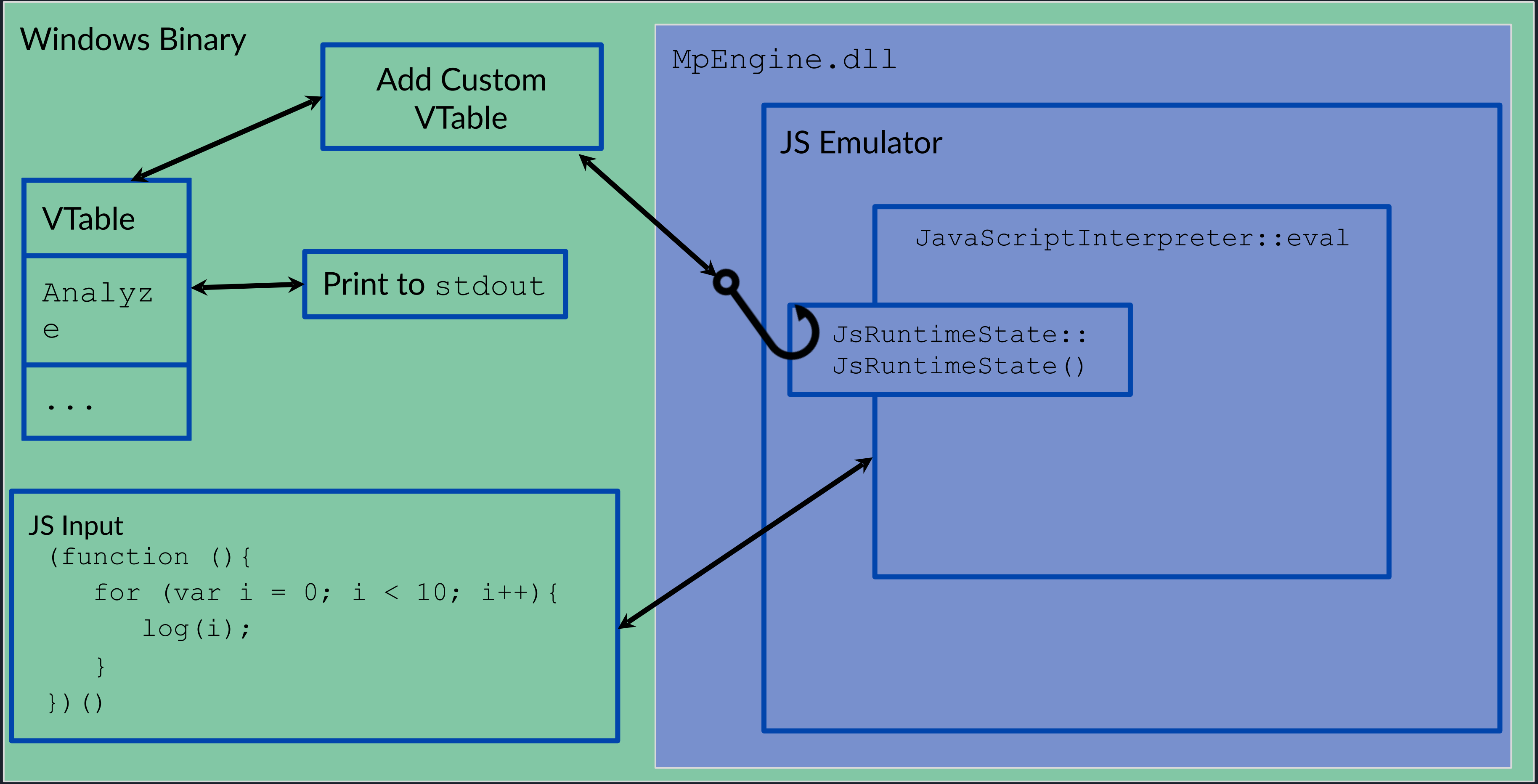
JavaScriptInterpreter::eval

JsRuntimeState::
JsRuntimeState()

JS Loader and Shell



JS Loader and Shell

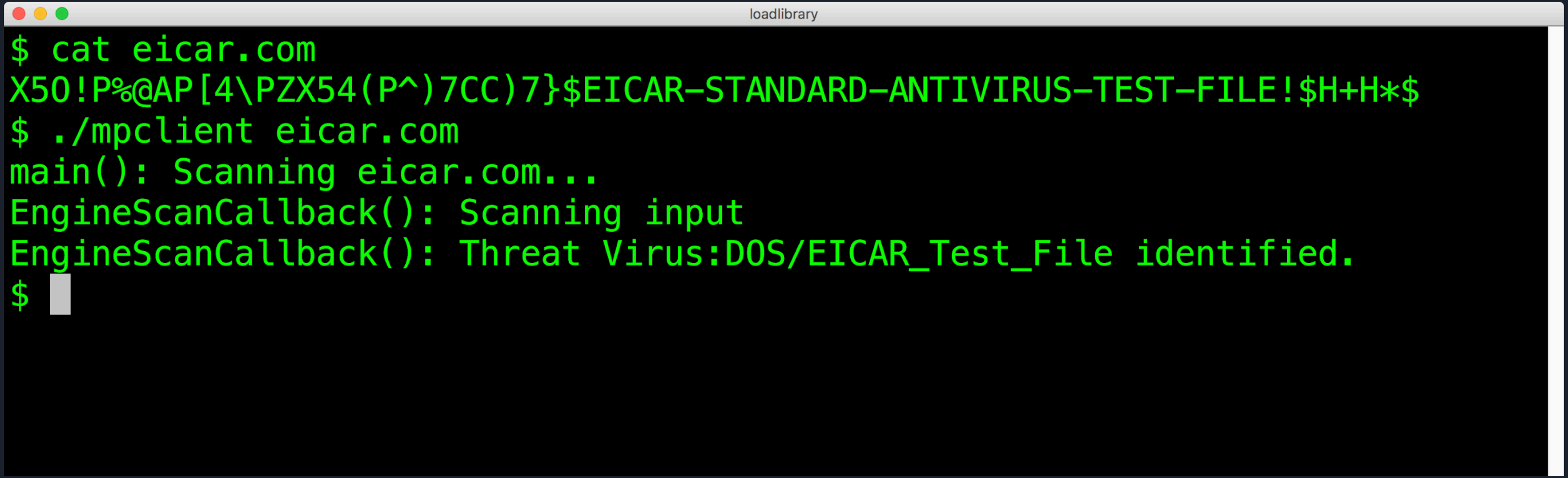


```
(function (){
   for (var i = 0; i < 10; i++){
      log(i);
   }
})()
```

Outline

1. Introduction
2. Tooling & Process
    a. Introduction
    b. JS Engine
    c. Emulator
3. Discussion
4. Conclusion

mpclient **Shell** git.io/fbp0X



```
$ cat eicar.com
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*$
$ ./mpclient eicar.com
main(): Scanning eicar.com...
EngineScanCallback(): Scanning input
EngineScanCallback(): Threat Virus:DOS/EICAR_Test_File identified.
$ 
```

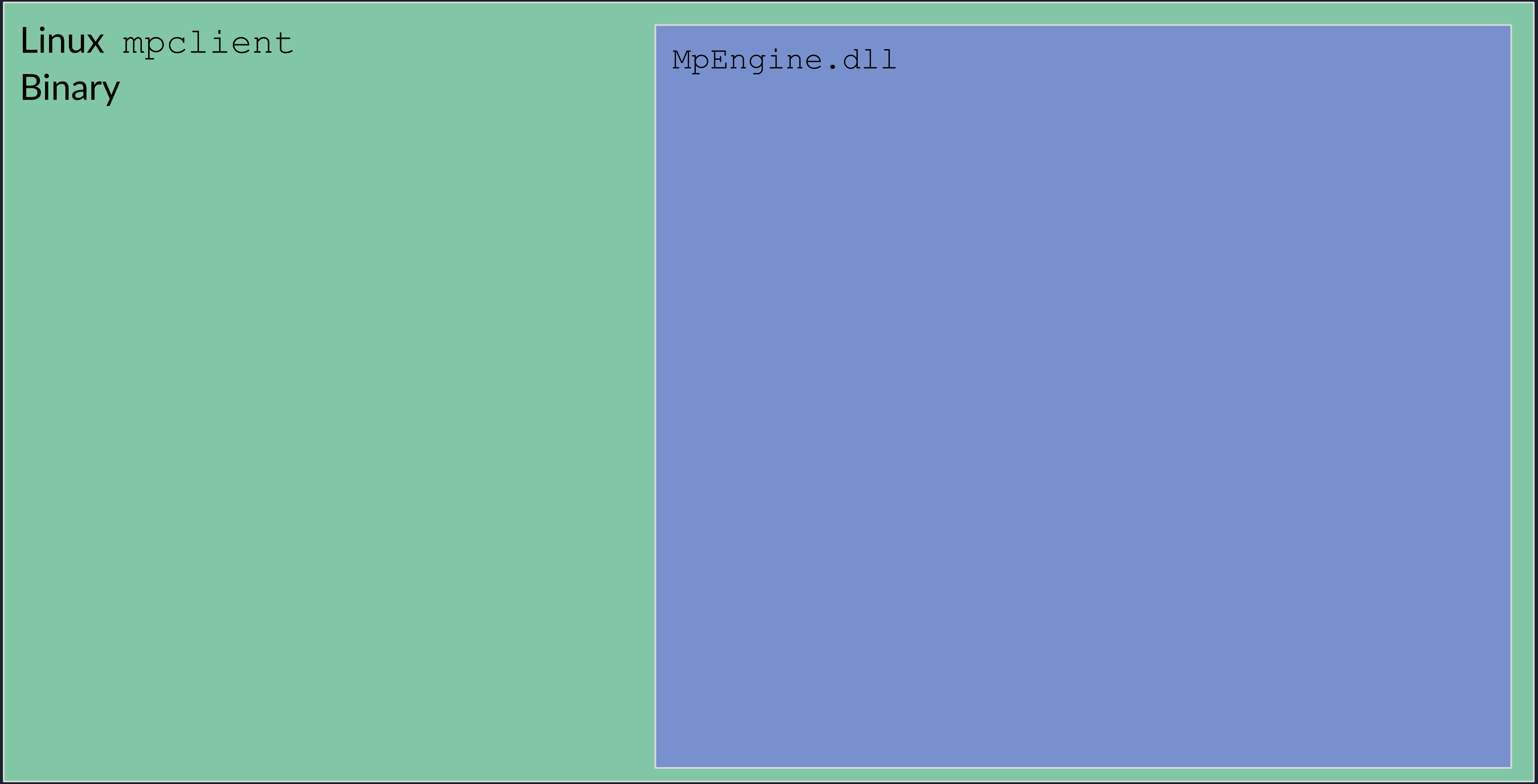
Tavis Ormandy’s open source tool

mpclient git.io/fbp0X

Linux `mpclient`

Binary

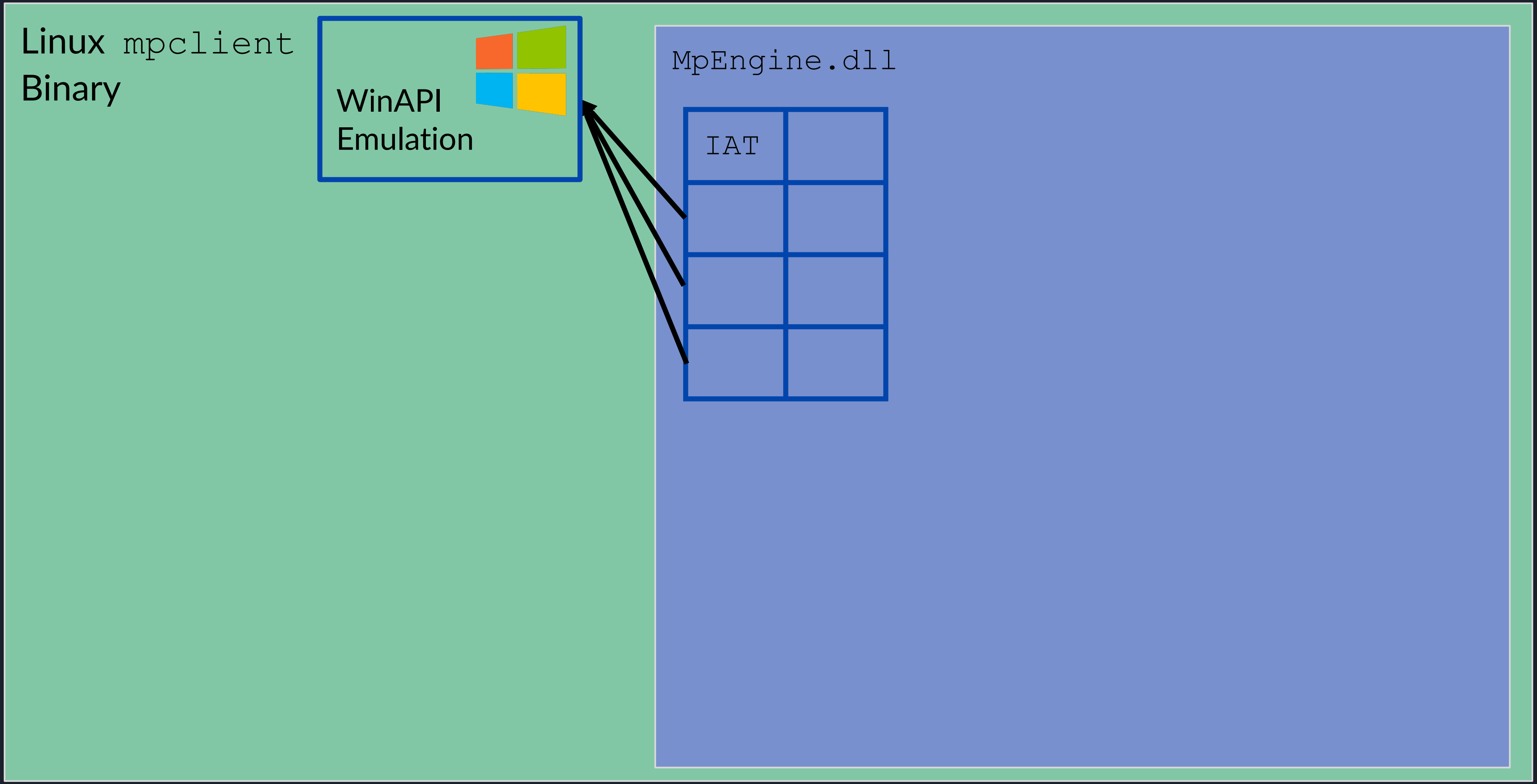
mpclient git.io/fbp0X



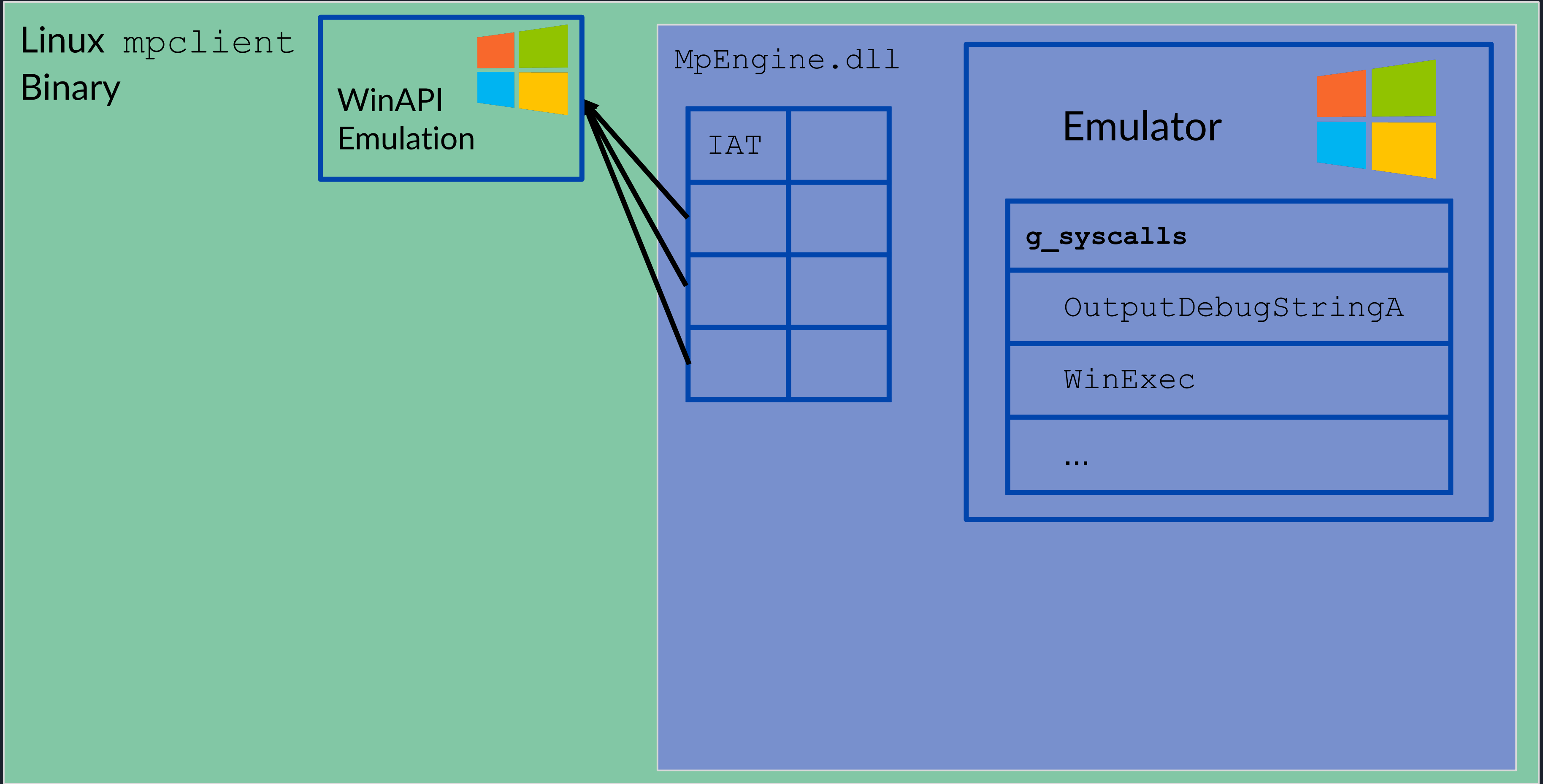
Linux `mpclient` Binary

`MpEngine.dll`

mpclient git.io/fbp0X



mpclient git.io/fbp0X



Linux mpclient Binary

WinAPI Emulation

MpEngine.dll

IAT

Emulator

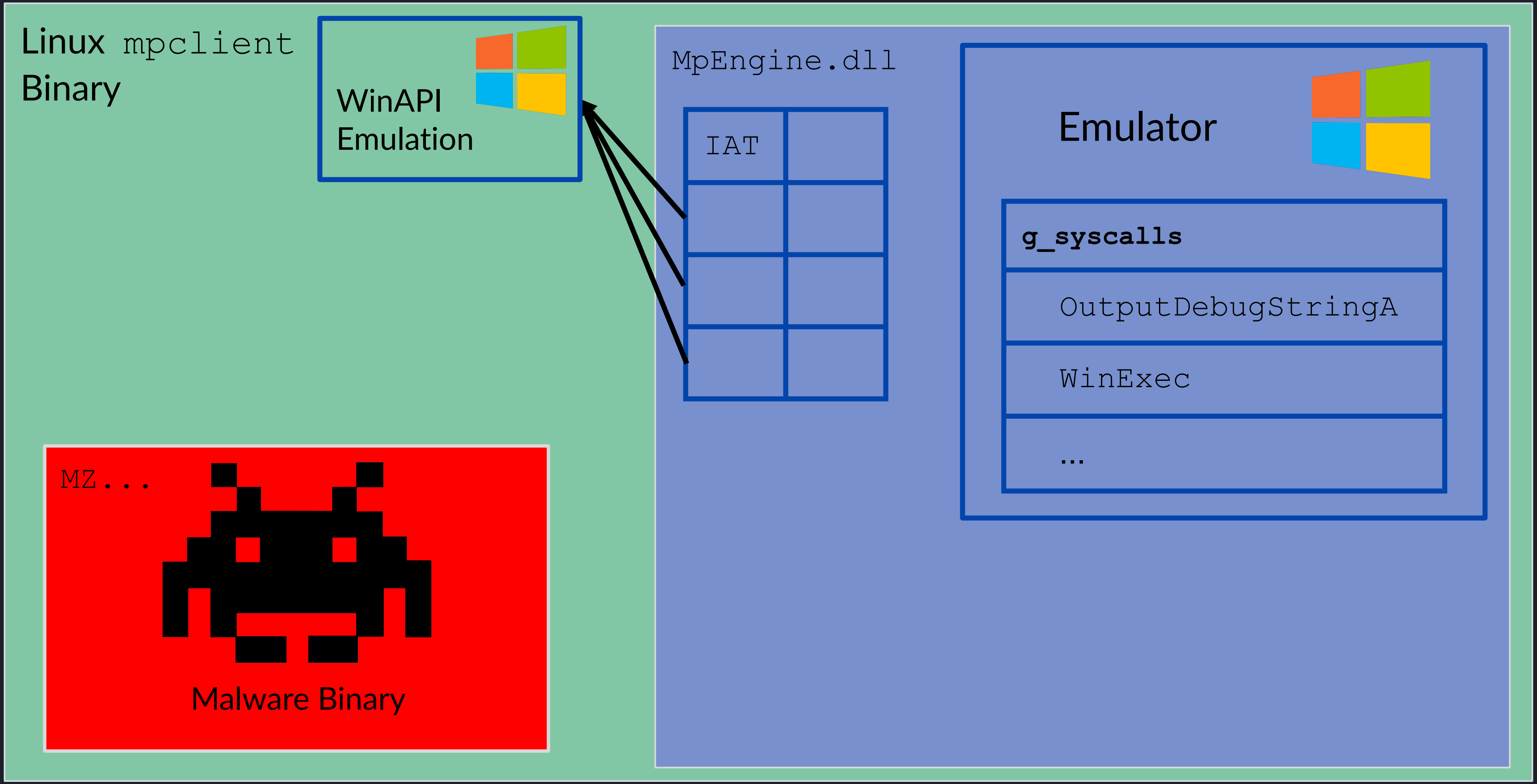
g_syscalls

OutputDebugStringA

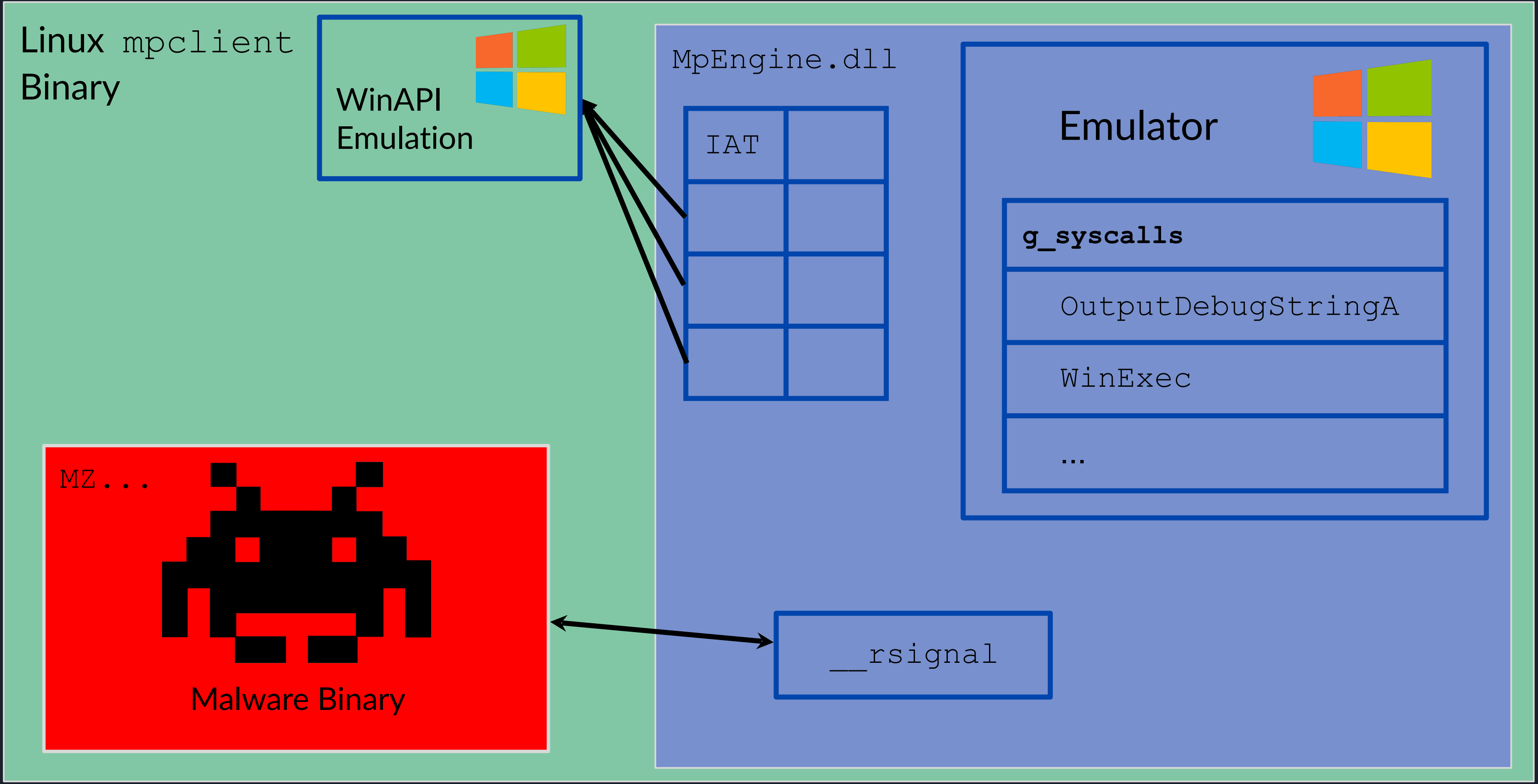
WinExec

...

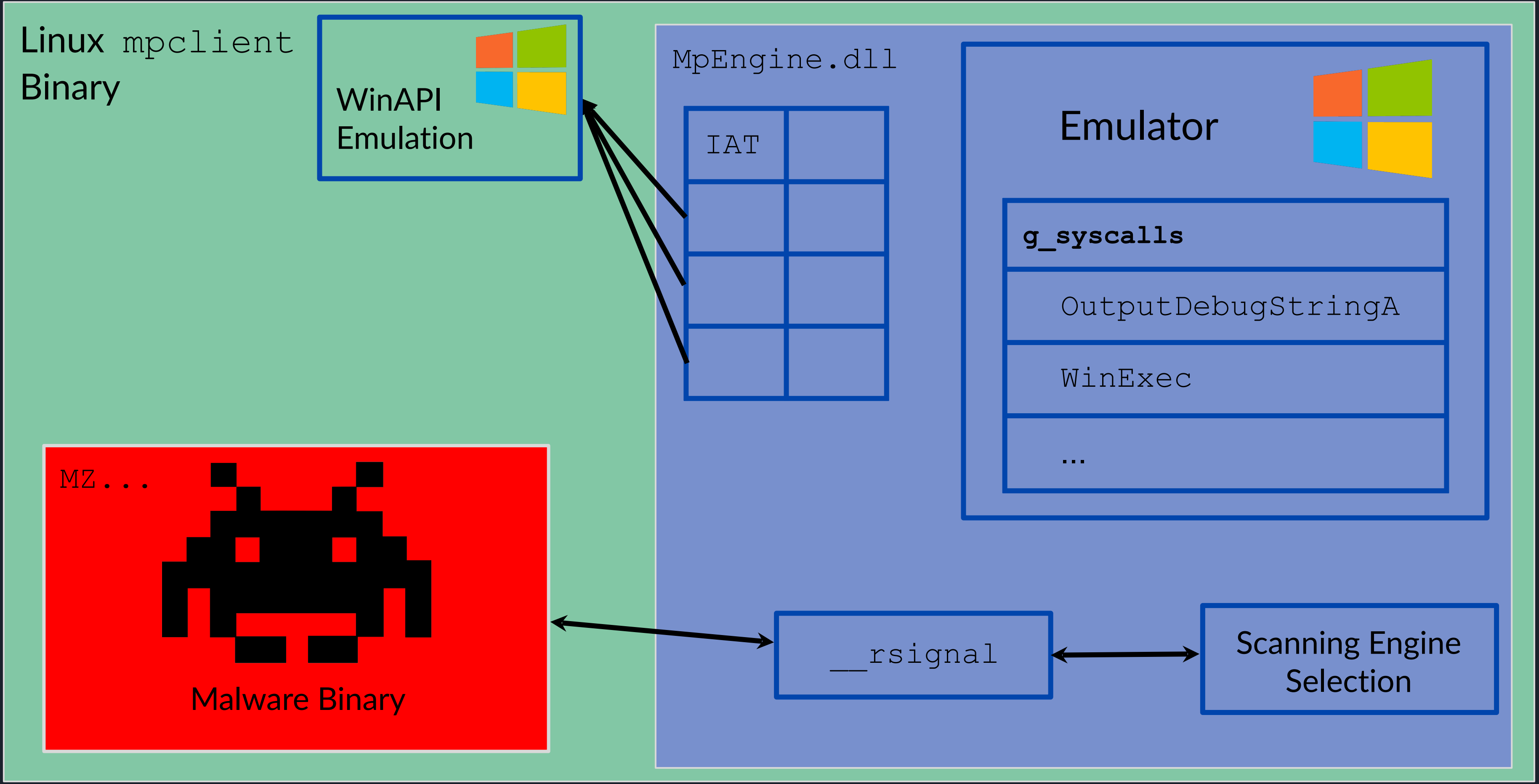
mpclient git.io/fbp0X



mpclient git.io/fbp0X



mpclient git.io/fbp0X



Linux `mpclient` Binary

WinAPI Emulation

`MpEngine.dll`

IAT

Emulator

g_syscalls

OutputDebugStringA

WinExec

...

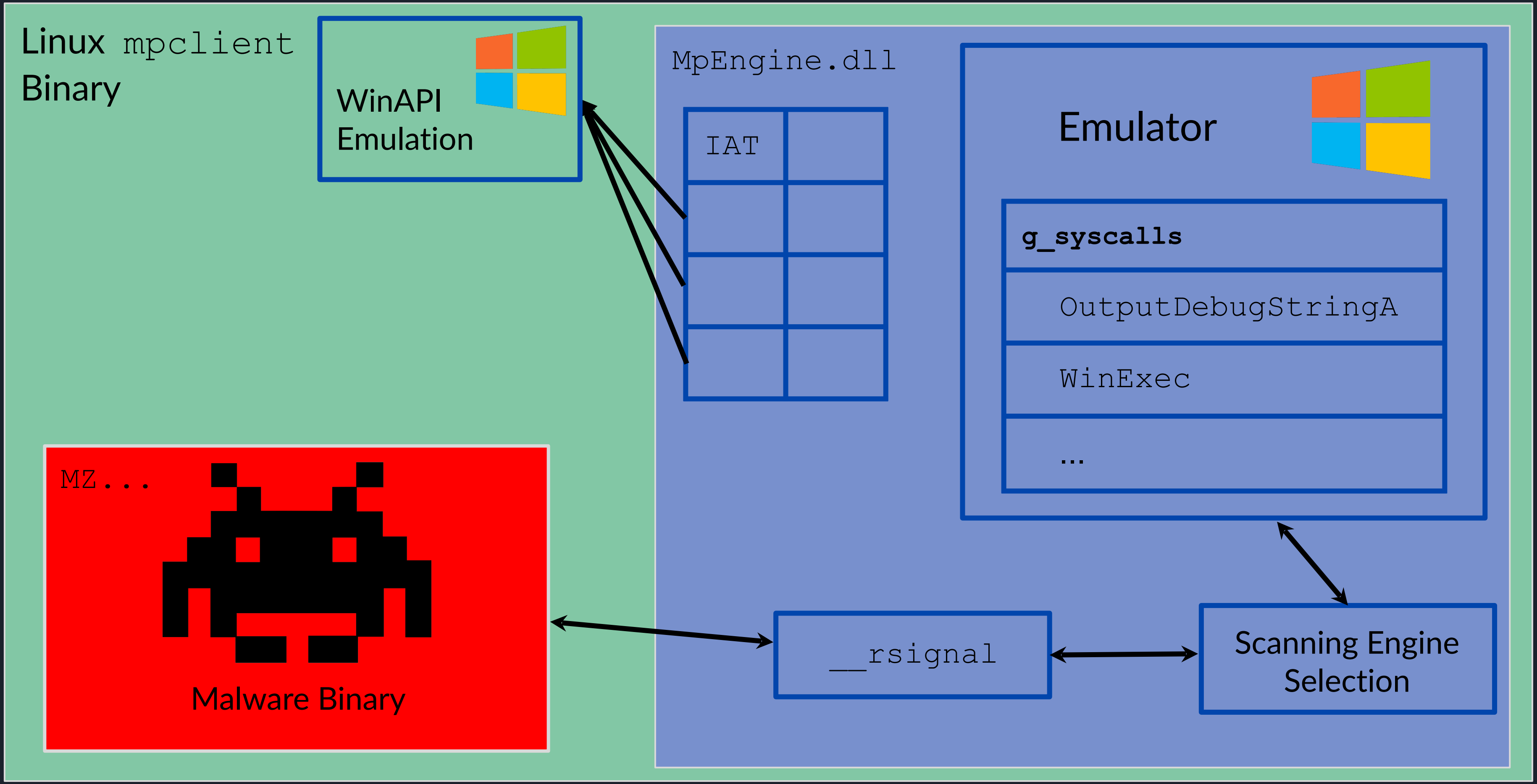
MZ...

Malware Binary

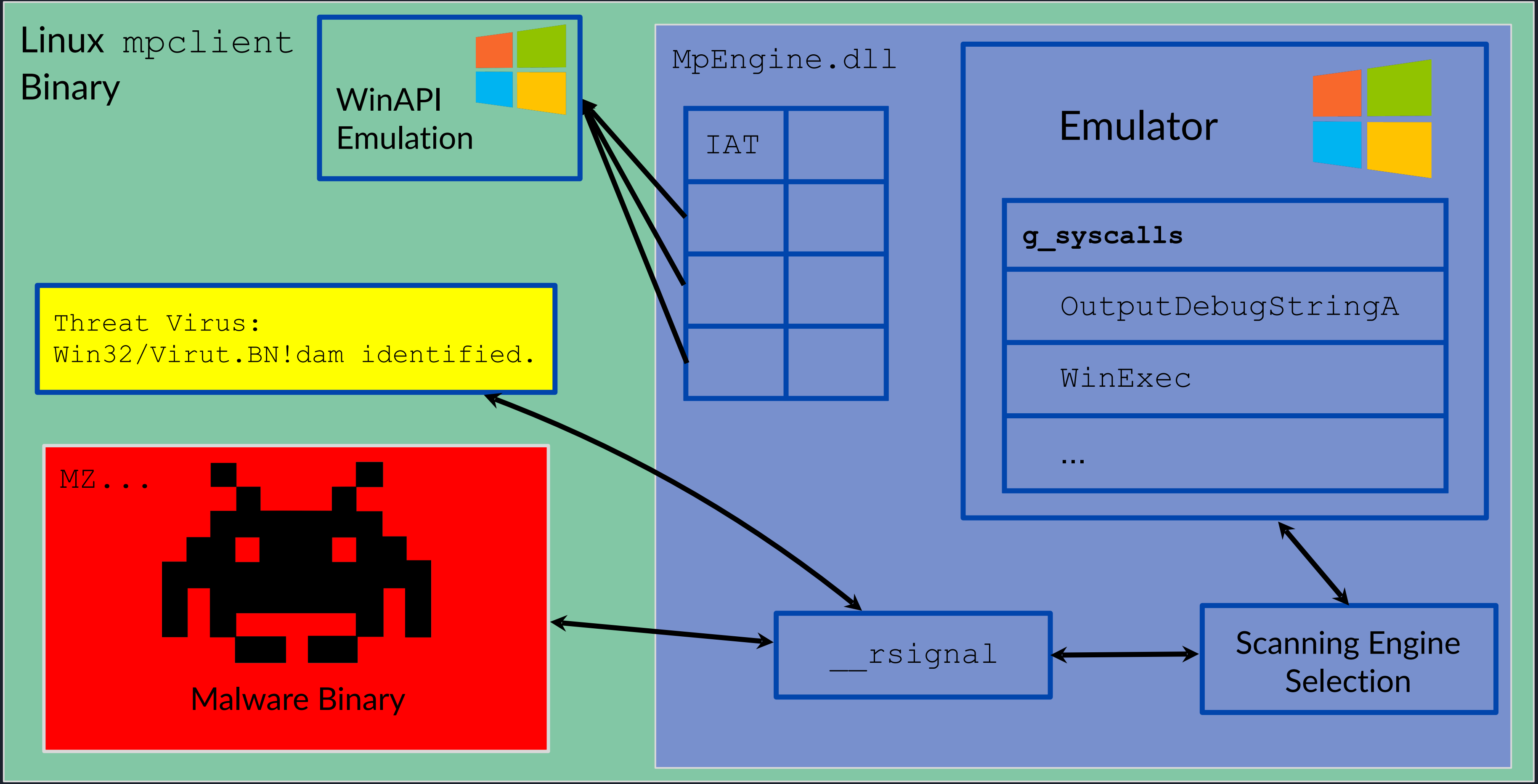
__rsignal

Scanning Engine Selection

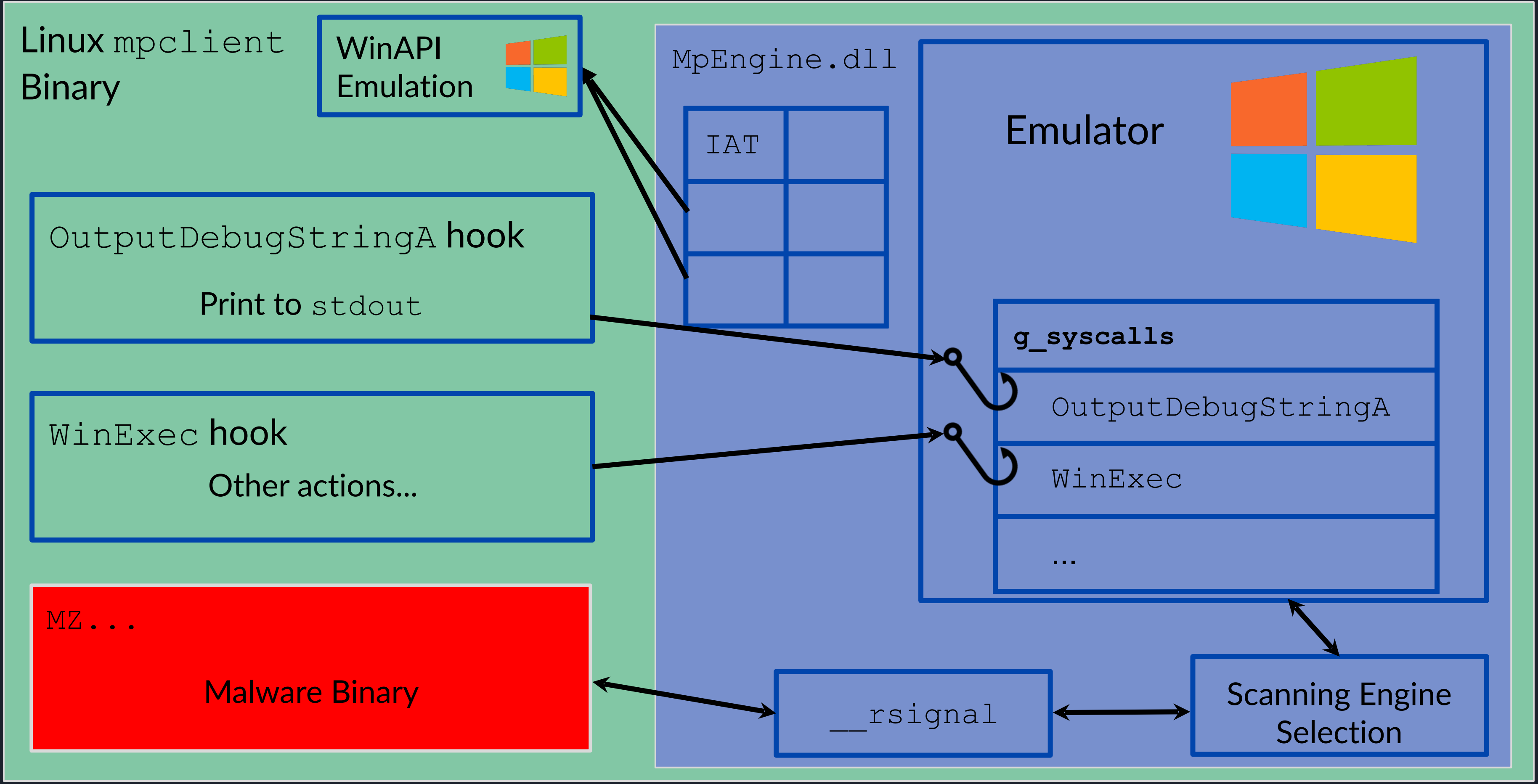
mpclient git.io/fbp0X



mpclient git.io/fbp0X



Modified `mpclient` - ~3k LoC added `github.com/0xAlexei`



Linux `mpclient` Binary

WinAPI Emulation

`OutputDebugStringA` hook

Print to `stdout`

`WinExec` hook

Other actions...

`MZ...`

Malware Binary

`MpEngine.dll`

`IAT`

Emulator

g_syscalls

`OutputDebugStringA`

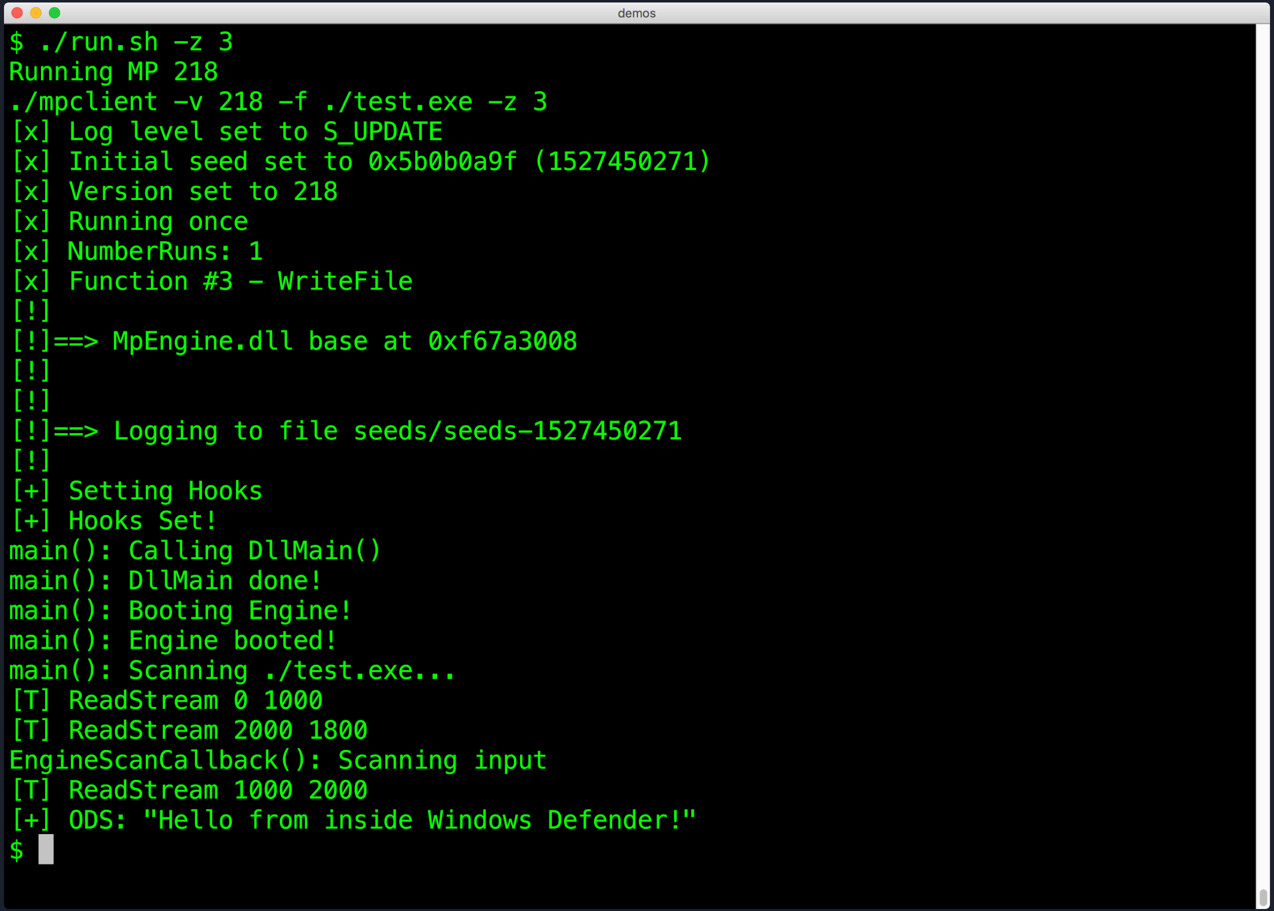
`WinExec`

...

`__rsignal`

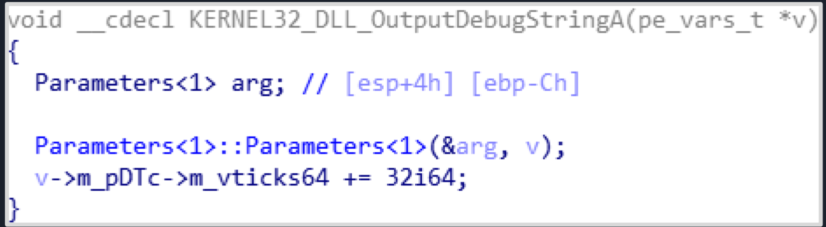
Scanning Engine Selection

Modified `mpclient`



```
$ ./run.sh -z 3
Running MP 218
./mpclient -v 218 -f ./test.exe -z 3
[x] Log level set to S_UPDATE
[x] Initial seed set to 0x5b0b0a9f (1527450271)
[x] Version set to 218
[x] Running once
[x] NumberRuns: 1
[x] Function #3 - WriteFile
[!]
[!]==> MpEngine.dll base at 0xf67a3008
[!]
[!]
[!]==> Logging to file seeds/seeds-1527450271
[!]
[+] Setting Hooks
[+] Hooks Set!
main(): Calling DllMain()
main(): DllMain done!
main(): Booting Engine!
main(): Engine booted!
main(): Scanning ./test.exe...
[T] ReadStream 0 1000
[T] ReadStream 2000 1800
EngineScanCallback(): Scanning input
[T] ReadStream 1000 2000
[+] ODS: "Hello from inside Windows Defender!"
$
```

OutputDebugStringA Hook

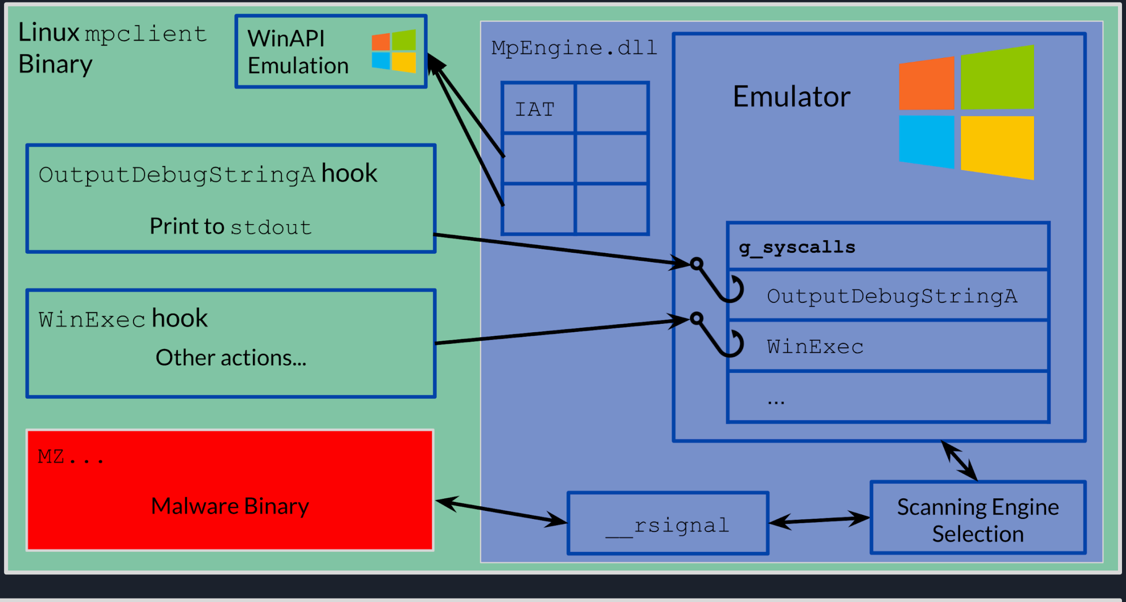


```
void __cdecl KERNEL32_DLL_OutputDebugStringA(pe_vars_t *v)
{
  Parameters<1> arg; // [esp+4h] [ebp-Ch]

  Parameters<1>::Parameters<1>(&arg, v);
  v->m_pDTc->m_vticks64 += 32i64;
}
```

Hook the native function pointer that gets called when `OutputDebugStringA` is called in-emulator

Use existing functions in Defender to interact with function parameters and virtual memory

Mark - Thanks for the idea!



Linux mpclient Binary

WinAPI Emulation

OutputDebugStringA hook

Print to stdout

WinExec hook

Other actions...

MZ...

Malware Binary

MpEngine.dll

IAT

Emulator

g_syscalls

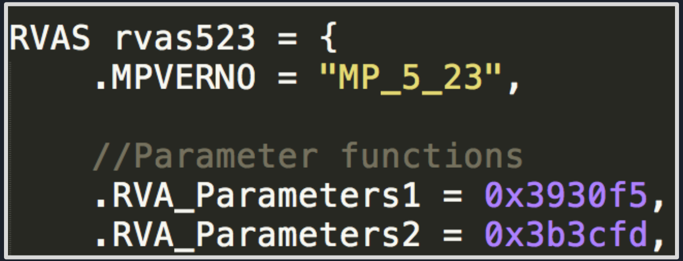
OutputDebugStringA

WinExec

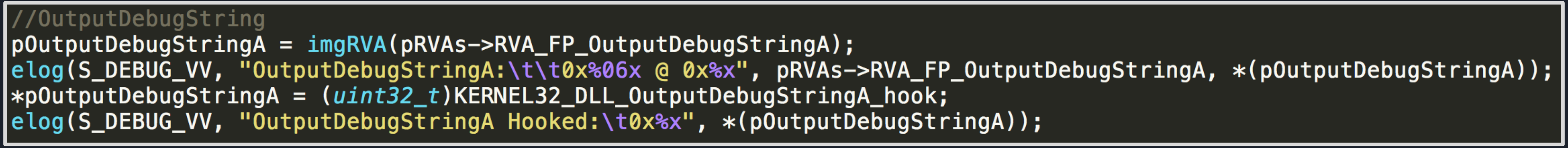
...

__rsignal

Scanning Engine Selection



```
RVAS rvas523 = {
    .MPVERNO = "MP_5_23",

    //Parameter functions
    .RVA_Parameters1 = 0x3930f5,
    .RVA_Parameters2 = 0x3b3cfd,
```



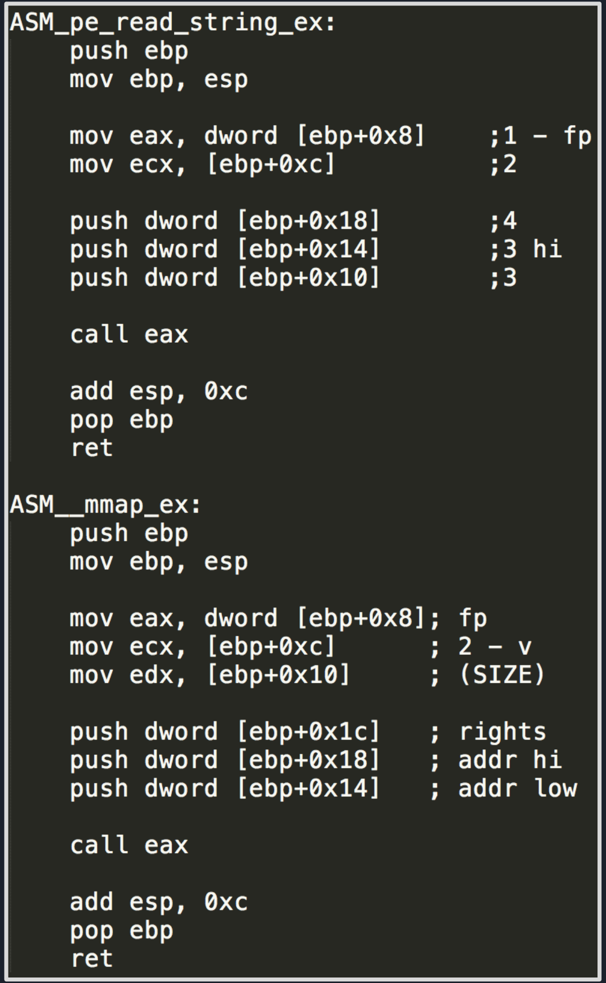
```
//OutputDebugString
pOutputDebugStringA = imgRVA(pRVAs->RVA_FP_OutputDebugStringA);
elog(S_DEBUG_VV, "OutputDebugStringA:\t\t0x%06x @ 0x%x", pRVAs->RVA_FP_OutputDebugStringA, *(pOutputDebugStringA));
*pOutputDebugStringA = (uint32_t)KERNEL32_DLL_OutputDebugStringA_hook;
elog(S_DEBUG_VV, "OutputDebugStringA Hooked:\t0x%x", *(pOutputDebugStringA));
```

Dealing With Calling Conventions

When calling `mpengine.dll` functions from `mpclient`: Difficulty of interoperability between MSVC and GCC compiled code

- Possible to massage compiler with `__attribute__` annotations

Easier solution - just hand-write assembly thunks to marshall arguments into the correct format



```
ASM_pe_read_string_ex:
    push ebp
    mov ebp, esp

    mov eax, dword [ebp+0x8]    ;1 – fp
    mov ecx, [ebp+0xc]          ;2

    push dword [ebp+0x18]       ;4
    push dword [ebp+0x14]       ;3 hi
    push dword [ebp+0x10]       ;3

    call eax

    add esp, 0xc
    pop ebp
    ret

ASM__mmap_ex:
    push ebp
    mov ebp, esp

    mov eax, dword [ebp+0x8]; fp
    mov ecx, [ebp+0xc]      ; 2 – v
    mov edx, [ebp+0x10]     ; (SIZE)

    push dword [ebp+0x1c]   ; rights
    push dword [ebp+0x18]   ; addr hi
    push dword [ebp+0x14]   ; addr low

    call eax

    add esp, 0xc
    pop ebp
    ret
```

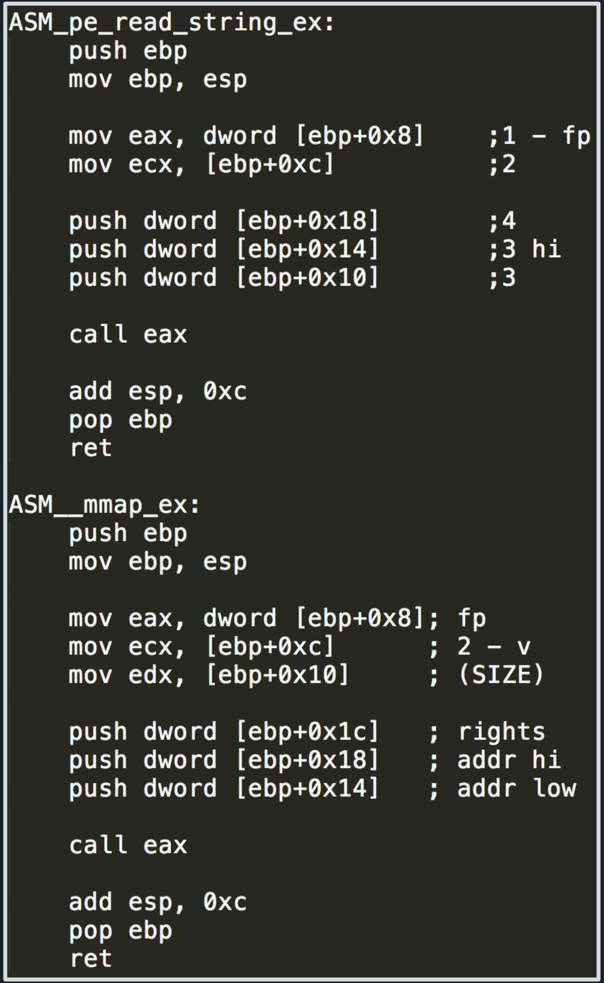
Dealing With Calling Conventions

When calling `mpengine.dll` functions from `mpclient`: Difficulty of interoperability between MSVC and GCC compiled code

- Possible to massage compiler with `__attribute__` annotations

Easier solution - just hand-write assembly thunks to marshall arguments into the correct format

```
BYTE * __fastcall __mmap_ex
(
    pe_vars_t * v,          // ecx
    unsigned int64 addr,    // too big for edx
    unsigned long  size,    // edx
    unsigned long  rights
);
```



```
ASM_pe_read_string_ex:
    push ebp
    mov ebp, esp

    mov eax, dword [ebp+0x8]     ;1 - fp
    mov ecx, [ebp+0xc]           ;2

    push dword [ebp+0x18]        ;4
    push dword [ebp+0x14]        ;3 hi
    push dword [ebp+0x10]        ;3

    call eax

    add esp, 0xc
    pop ebp
    ret

ASM__mmap_ex:
    push ebp
    mov ebp, esp

    mov eax, dword [ebp+0x8]; fp
    mov ecx, [ebp+0xc]       ; 2 - v
    mov edx, [ebp+0x10]      ; (SIZE)

    push dword [ebp+0x1c]    ; rights
    push dword [ebp+0x18]    ; addr hi
    push dword [ebp+0x14]    ; addr low

    call eax

    add esp, 0xc
    pop ebp
    ret
```

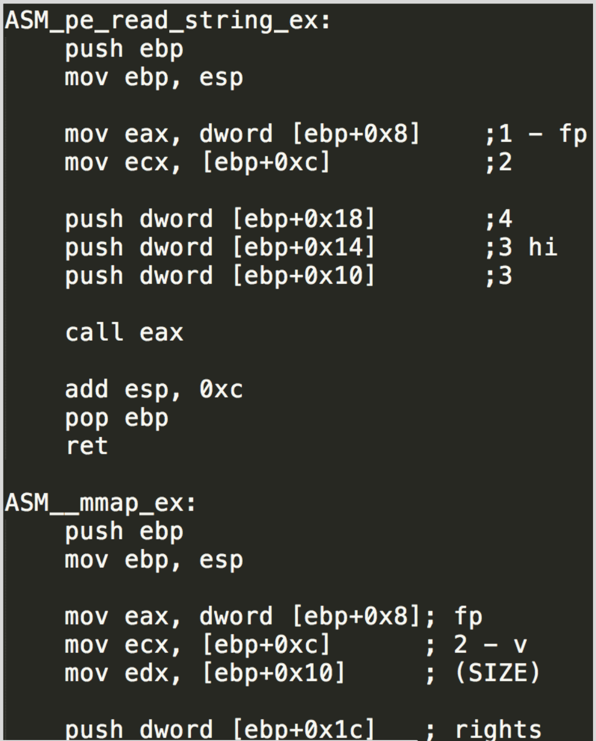
Dealing With Calling Conventions

When calling `mpengine.dll` functions from `mpclient`: Difficulty of interoperability between MSVC and GCC compiled code

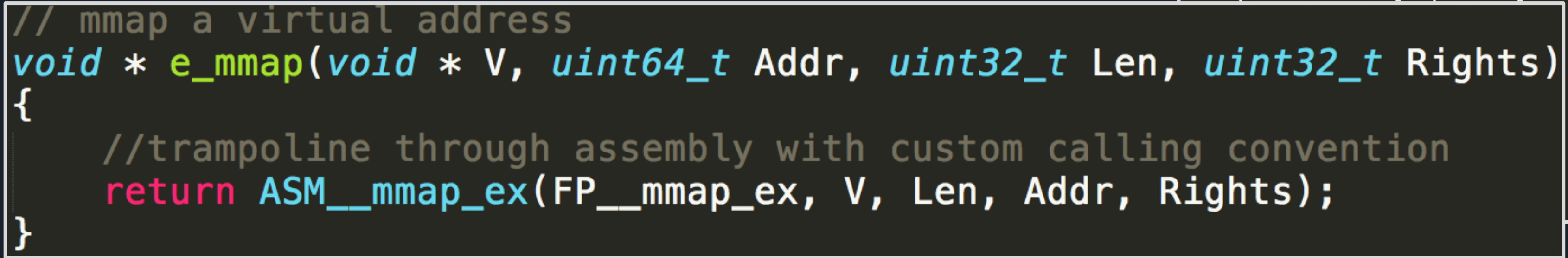
- Possible to massage compiler with `__attribute__` annotations

Easier solution - just hand-write assembly thunks to marshall arguments into the correct format

```
BYTE * __fastcall __mmap_ex
(
    pe_vars_t * v,          // ecx
    unsigned int64 addr,    // too big for edx
    unsigned long  size,    // edx
    unsigned long  rights
);
```



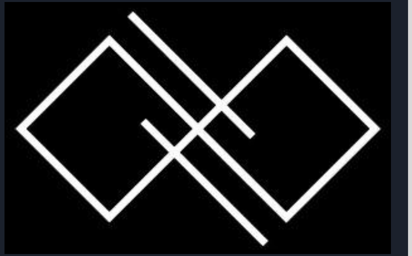
```
ASM_pe_read_string_ex:
    push ebp
    mov ebp, esp

    mov eax, dword [ebp+0x8]     ;1 - fp
    mov ecx, [ebp+0xc]           ;2

    push dword [ebp+0x18]        ;4
    push dword [ebp+0x14]        ;3 hi
    push dword [ebp+0x10]        ;3

    call eax

    add esp, 0xc
    pop ebp
    ret

ASM__mmap_ex:
    push ebp
    mov ebp, esp

    mov eax, dword [ebp+0x8]; fp
    mov ecx, [ebp+0xc]      ; 2 - v
    mov edx, [ebp+0x10]     ; (SIZE)

    push dword [ebp+0x1c]   ; rights
                            ; addr hi
                            ; addr low
```

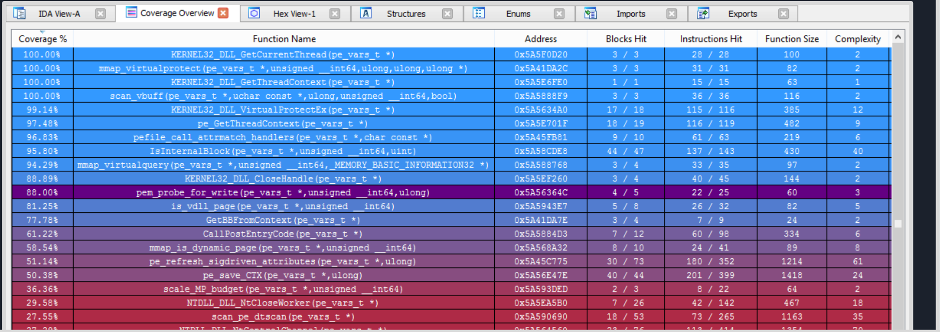


```
// mmap a virtual address
void * e_mmap(void * V, uint64_t Addr, uint32_t Len, uint32_t Rights)
{
    //trampoline through assembly with custom calling convention
    return ASM__mmap_ex(FP__mmap_ex, V, Len, Addr, Rights);
}
```

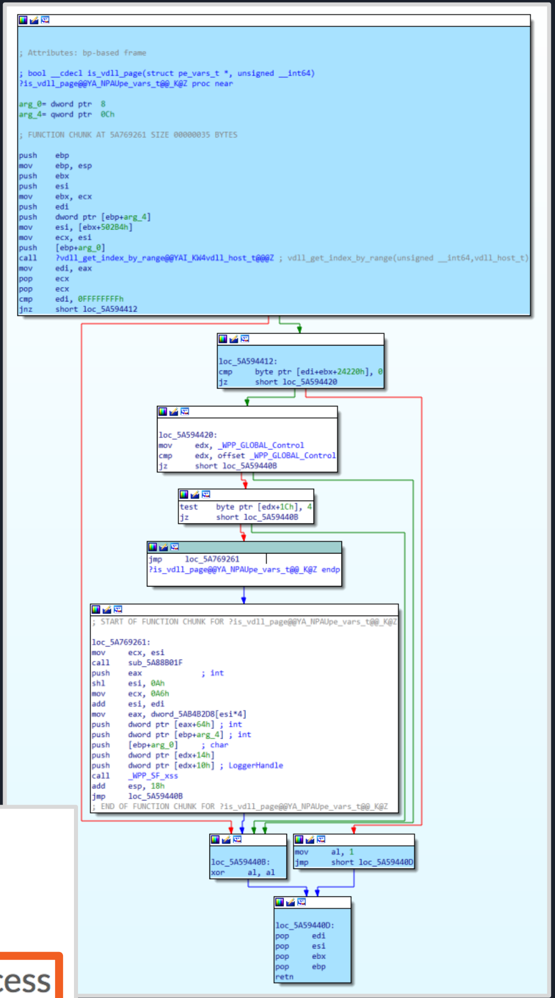
Dynamic Analysis - Code Coverage

- Getting an overview of what subsystems are being hit is helpful in characterizing a scan or emulation session
    - Breakpoints are too granular
- Emulator has no output other than malware identification
- Lighthouse code coverage plugin for IDA Pro from Markus Gaasedelen of Ret2 Systems / RPISEC





Coverage %	Function Name	Address	Blocks Hit	Instructions Hit	Function Size	Complexity
100.00%	KERNEL32_DLL_GetCurrentThread(pe_vars_t *)	0x5A5F0D20	3 / 3	28 / 28	100	2
100.00%	mmap_virtualprotect(pe_vars_t *,unsigned __int64,ulong,ulong,ulong *)	0x5A41DA2C	3 / 3	31 / 31	82	2
100.00%	KERNEL32_DLL_GetThreadContext(pe_vars_t *)	0x5A5E6FE0	1 / 1	15 / 15	63	1
100.00%	scan_vbuff(pe_vars_t *,uchar const *,ulong,unsigned __int64,bool)	0x5A5888F9	3 / 3	36 / 36	116	2
99.14%	KERNEL32_DLL_VirtualProtectEx(pe_vars_t *)	0x5A5634A0	17 / 18	115 / 116	385	12
97.48%	pe_GetThreadContext(pe_vars_t *)	0x5A5E701F	18 / 19	116 / 119	482	9
96.83%	pefile_call_attrmatch_handlers(pe_vars_t *,char const *)	0x5A45FB81	9 / 10	61 / 63	219	6
95.80%	IsInternalBlock(pe_vars_t *,unsigned __int64,uint)	0x5A58CDE8	44 / 47	137 / 143	430	40
94.29%	mmap_virtualquery(pe_vars_t *,unsigned __int64,_MEMORY_BASIC_INFORMATION32 *)	0x5A588768	3 / 4	33 / 35	97	2
88.89%	KERNEL32_DLL_CloseHandle(pe_vars_t *)	0x5A5EF260	3 / 4	40 / 45	144	2
88.00%	pem_probe_for_write(pe_vars_t *,unsigned __int64,ulong)	0x5A56364C	4 / 5	22 / 25	60	3
81.25%	is_vdll_page(pe_vars_t *,unsigned __int64)	0x5A5943E7	5 / 8	26 / 32	82	5
77.78%	GetBBFromContext(pe_vars_t *)	0x5A41DA7E	3 / 4	7 / 9	24	2
61.22%	CallPostEntryCode(pe_vars_t *)	0x5A5884D3	7 / 12	60 / 98	334	6
58.54%	mmap_is_dynamic_page(pe_vars_t *,unsigned __int64)	0x5A568A32	8 / 10	24 / 41	89	8
51.14%	pe_refresh_sigdriven_attributes(pe_vars_t *,ulong)	0x5A45C775	30 / 73	180 / 352	1214	61
50.38%	pe_save_CTX(pe_vars_t *,ulong)	0x5A56E47E	40 / 44	201 / 399	1418	24
36.36%	scale_MP_budget(pe_vars_t *,unsigned __int64)	0x5A593DED	2 / 3	8 / 22	64	2
29.58%	NTDLL_DLL_NtCloseWorker(pe_vars_t *)	0x5A5EA5B0	7 / 26	42 / 142	467	18
27.55%	scan_pe_dtscan(pe_vars_t *)	0x5A590690	18 / 53	73 / 265	1163	35



```
; Attributes: bp-based frame

; bool __cdecl is_vdll_page(struct pe_vars_t *, unsigned __int64)
?is_vdll_page@@YA_NPAUpe_vars_t@@_K@Z proc near

arg_0= dword ptr  8
arg_4= qword ptr  0Ch

; FUNCTION CHUNK AT 5A769261 SIZE 00000035 BYTES

push    ebp
mov     ebp, esp
push    ebx
push    esi
mov     ebx, ecx
push    edi
push    dword ptr [ebp+arg_4]
mov     esi, [ebx+502B4h]
mov     ecx, esi
push    [ebp+arg_0]
call    ?vdll_get_index_by_range@@YAI_KW4vdll_host_t@@@Z ; vdll_get_index_by_range(unsigned __int64,vdll_host_t)
mov     edi, eax
pop     ecx
pop     ecx
cmp     edi, 0FFFFFFFFh
jnz     short loc_5A594412
```

```
loc_5A594412:
cmp     byte ptr [edi+ebx+24220h], 0
jz      short loc_5A594420
```

```
loc_5A594420:
mov     edx, _WPP_GLOBAL_Control
cmp     edx, offset _WPP_GLOBAL_Control
jz      short loc_5A59440B
```

```
test    byte ptr [edx+1Ch], 4
jz      short loc_5A59440B
```

```
jmp     loc_5A769261
?is_vdll_page@@YA_NPAUpe_vars_t@@_K@Z endp
```

```
; START OF FUNCTION CHUNK FOR ?is_vdll_page@@YA_NPAUpe_vars_t@@_K@Z

loc_5A769261:
mov     ecx, esi
call    sub_5A88B01F
push    eax             ; int
shl     esi, 0Ah
mov     ecx, 0A6h
add     esi, edi
mov     eax, dword_5AB4B2D8[esi*4]
push    dword ptr [eax+64h] ; int
push    dword ptr [ebp+arg_4] ; int
push    [ebp+arg_0]     ; char
push    dword ptr [edx+14h]
push    dword ptr [edx+10h] ; LoggerHandle
call    _WPP_SF_xss
add     esp, 18h
jmp     loc_5A59440B
; END OF FUNCTION CHUNK FOR ?is_vdll_page@@YA_NPAUpe_vars_t@@_K@Z
```

```
loc_5A59440B:
xor     al, al
```

```
mov     al, 1
jmp     short loc_5A59440D
```

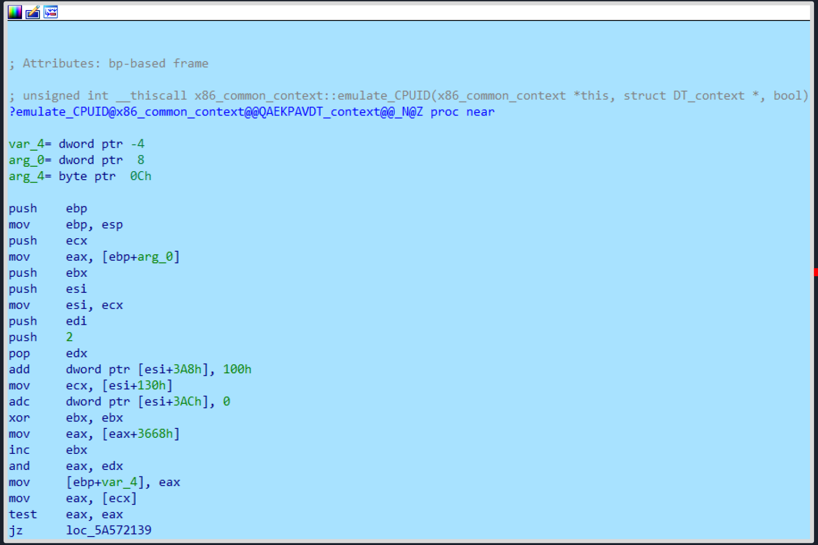
```
loc_5A59440D:
pop     edi
pop     esi
pop     ebx
pop     ebp
retn
```

Examples:

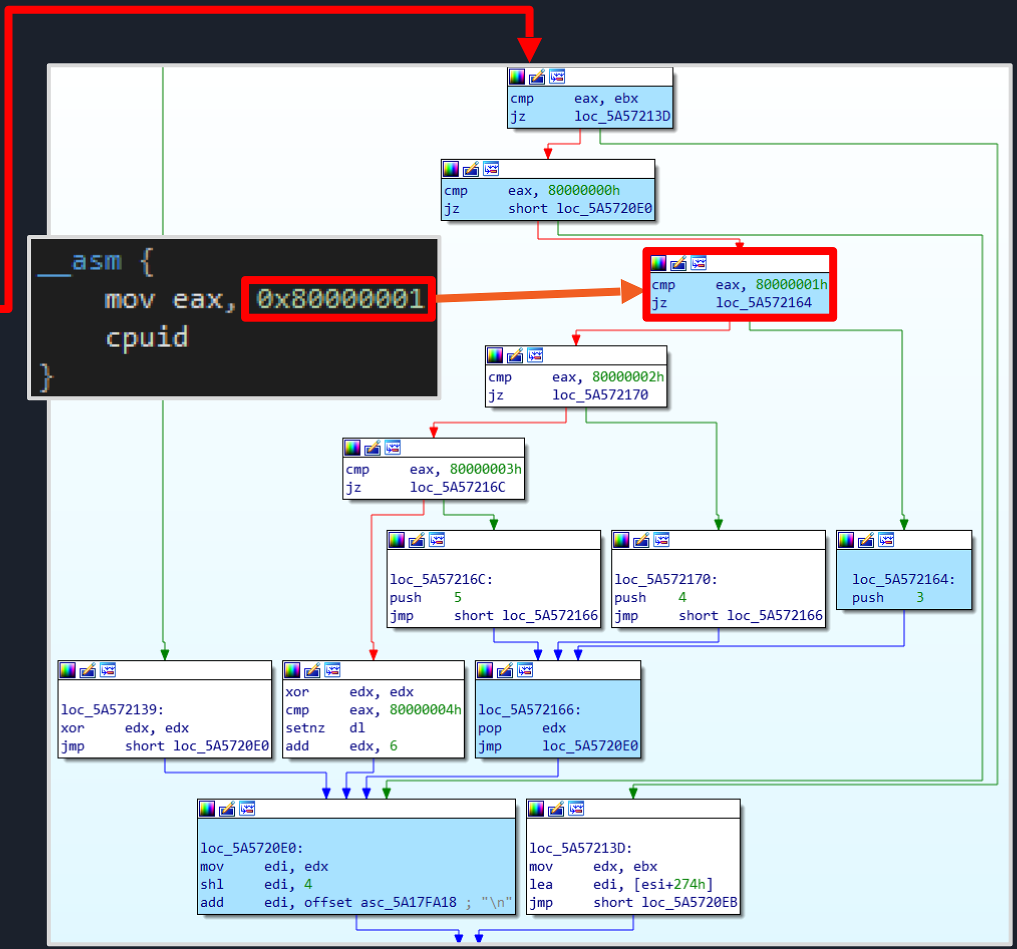
Halvar Flake’s SSTIC 2018 keynote

- Getting coverage traces from MPENGINE.DLL - difficult because of privileged process

x86_common_context::emulate_CPUID



```
; Attributes: bp-based frame

; unsigned int __thiscall x86_common_context::emulate_CPUID(x86_common_context *this, struct DT_context *, bool)
?emulate_CPUID@x86_common_context@@QAEKPAVDT_context@@_N@Z proc near

var_4= dword ptr -4
arg_0= dword ptr  8
arg_4= byte ptr  0Ch

push    ebp
mov     ebp, esp
push    ecx
mov     eax, [ebp+arg_0]
push    ebx
push    esi
mov     esi, ecx
push    edi
push    2
pop     edx
add     dword ptr [esi+3A8h], 100h
mov     ecx, [esi+130h]
adc     dword ptr [esi+3ACh], 0
xor     ebx, ebx
mov     eax, [eax+3668h]
inc     ebx
and     eax, edx
mov     [ebp+var_4], eax
mov     eax, [ecx]
test    eax, eax
jz      loc_5A572139
```



```
__asm {
    mov eax, 0x80000001
    cpuid
}
```

```
cmp     eax, ebx
jz      loc_5A57213D
```

```
cmp     eax, 80000000h
jz      short loc_5A5720E0
```

```
cmp     eax, 80000001h
jz      loc_5A572164
```

```
cmp     eax, 80000002h
jz      loc_5A572170
```

```
cmp     eax, 80000003h
jz      loc_5A57216C
```

```
loc_5A57216C:
push    5
jmp     short loc_5A572166
```

```
loc_5A572170:
push    4
jmp     short loc_5A572166
```

```
loc_5A572164:
push    3
```

```
loc_5A572139:
xor     edx, edx
jmp     short loc_5A5720E0
```

```
xor     edx, edx
cmp     eax, 80000004h
setnz   dl
add     edx, 6
```

```
loc_5A572166:
pop     edx
jmp     loc_5A5720E0
```

```
loc_5A5720E0:
mov     edi, edx
shl     edi, 4
add     edi, offset asc_5A17FA18 ; "\n"
```

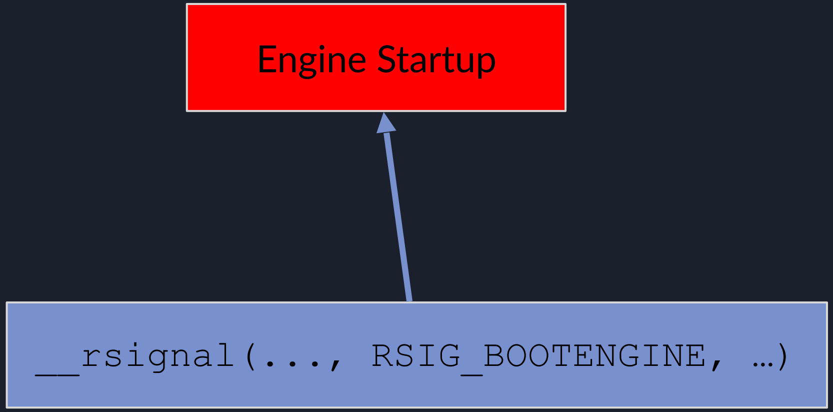
```
loc_5A57213D:
mov     edx, ebx
lea     edi, [esi+274h]
jmp     short loc_5A5720EB
```

Visualize emulator code coverage when emulating a given “malware” binary

Tracing Timeline

Pintool must be enlightened about custom loaded `mpengine.dll` location - take callback stub ideas from Tavis Ormandy's `deepcover` Pintool

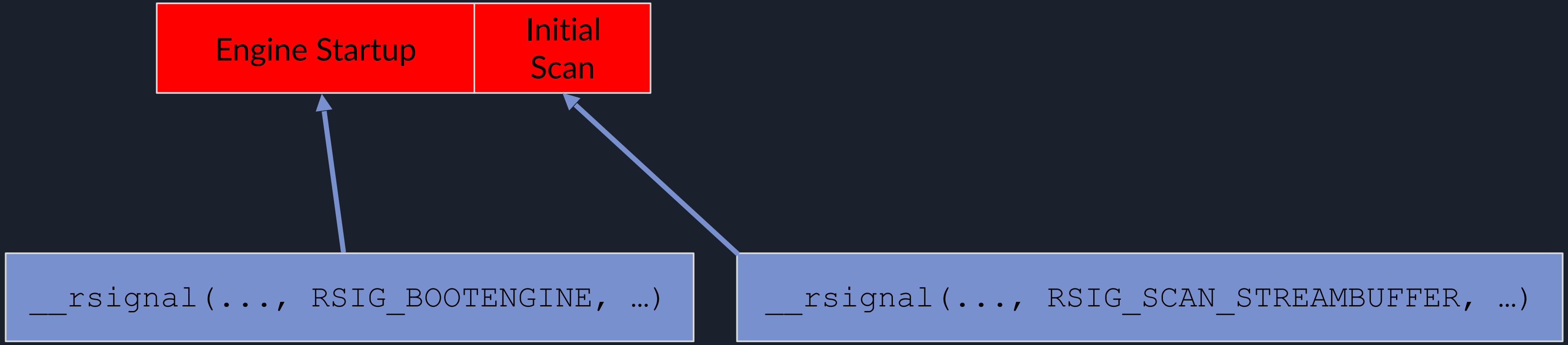
`github.com/taviso/loadlibrary/tree/master/coverage`



Tracing Timeline

Pintool must be enlightened about custom loaded `mpengine.dll` location - take callback stub ideas from Tavis Ormandy's `deepcover` Pintool

`github.com/taviso/loadlibrary/tree/master/coverage`



Engine Startup

Initial Scan

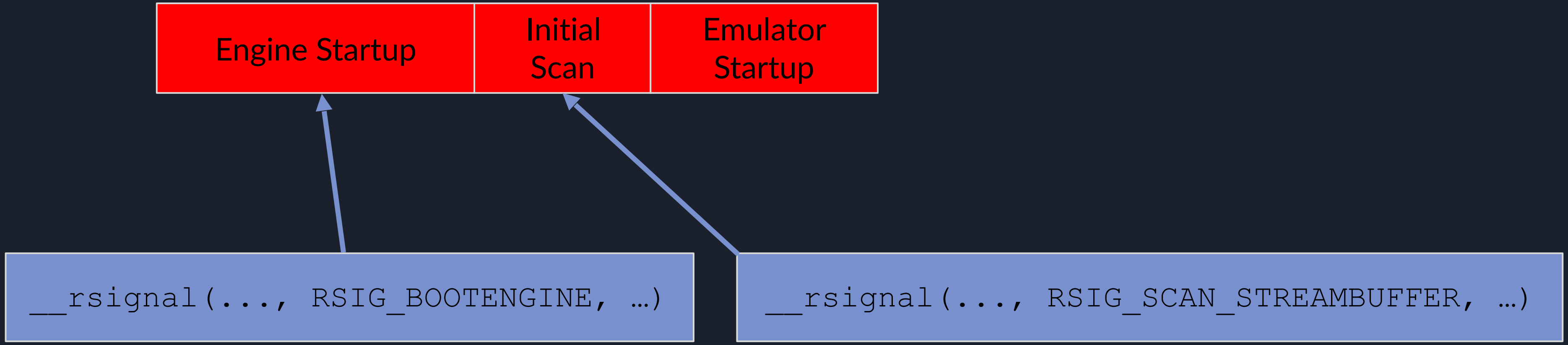
`__rsignal(..., RSIG_BOOTENGINE, …)`

`__rsignal(..., RSIG_SCAN_STREAMBUFFER, …)`

Tracing Timeline

Pintool must be enlightened about custom loaded `mpengine.dll` location - take callback stub ideas from Tavis Ormandy's `deepcover` Pintool

`github.com/taviso/loadlibrary/tree/master/coverage`



Engine Startup

Initial Scan

Emulator Startup

`__rsignal(..., RSIG_BOOTENGINE, …)`

`__rsignal(..., RSIG_SCAN_STREAMBUFFER, …)`

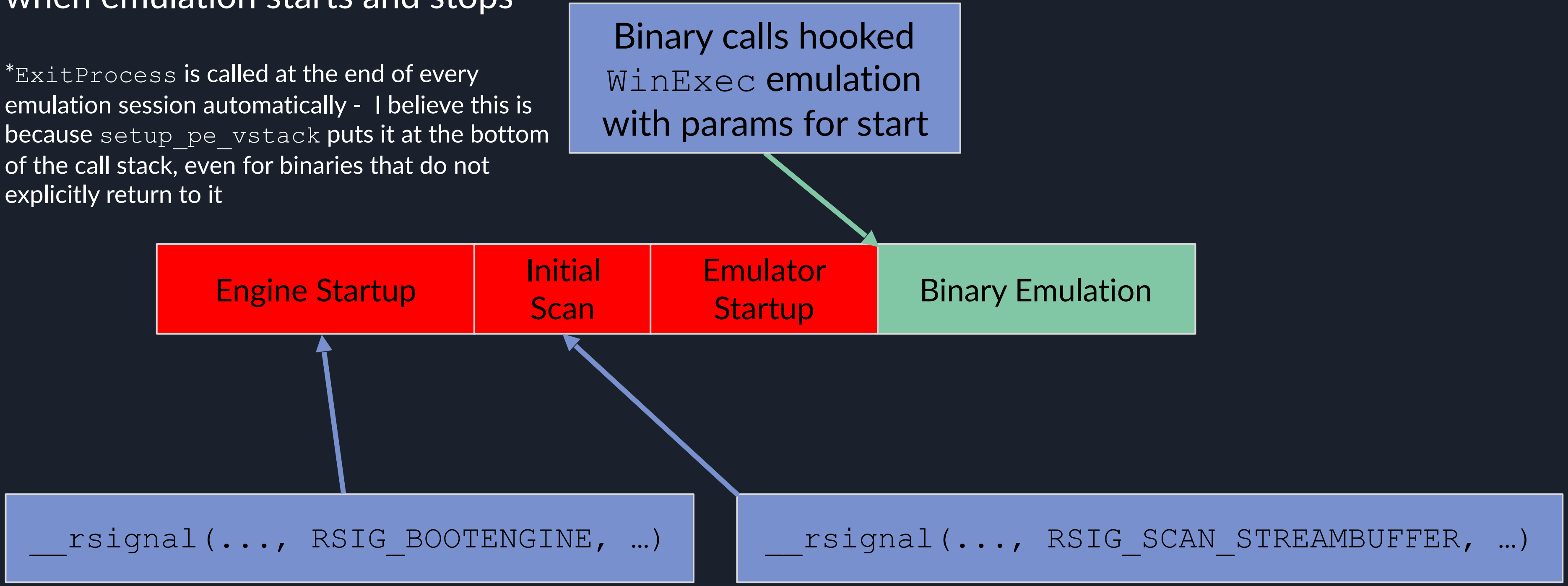
Tracing Timeline

Hooking Defender's emulation functions for WinExec and ExitProcess allows us to know when emulation starts and stops*

*ExitProcess is called at the end of every emulation session automatically - I believe this is because setup_pe_vstack puts it at the bottom of the call stack, even for binaries that do not explicitly return to it

Pintool must be enlightened about custom loaded mpengine.dll location - take callback stub ideas from Tavis Ormandy's deepcover Pintool

github.com/taviso/loadlibrary/tree/master/coverage



Binary calls hooked WinExec emulation with params for start

Engine Startup

Initial Scan

Emulator Startup

Binary Emulation

__rsignal(..., RSIG_BOOTENGINE, …)

__rsignal(..., RSIG_SCAN_STREAMBUFFER, …)

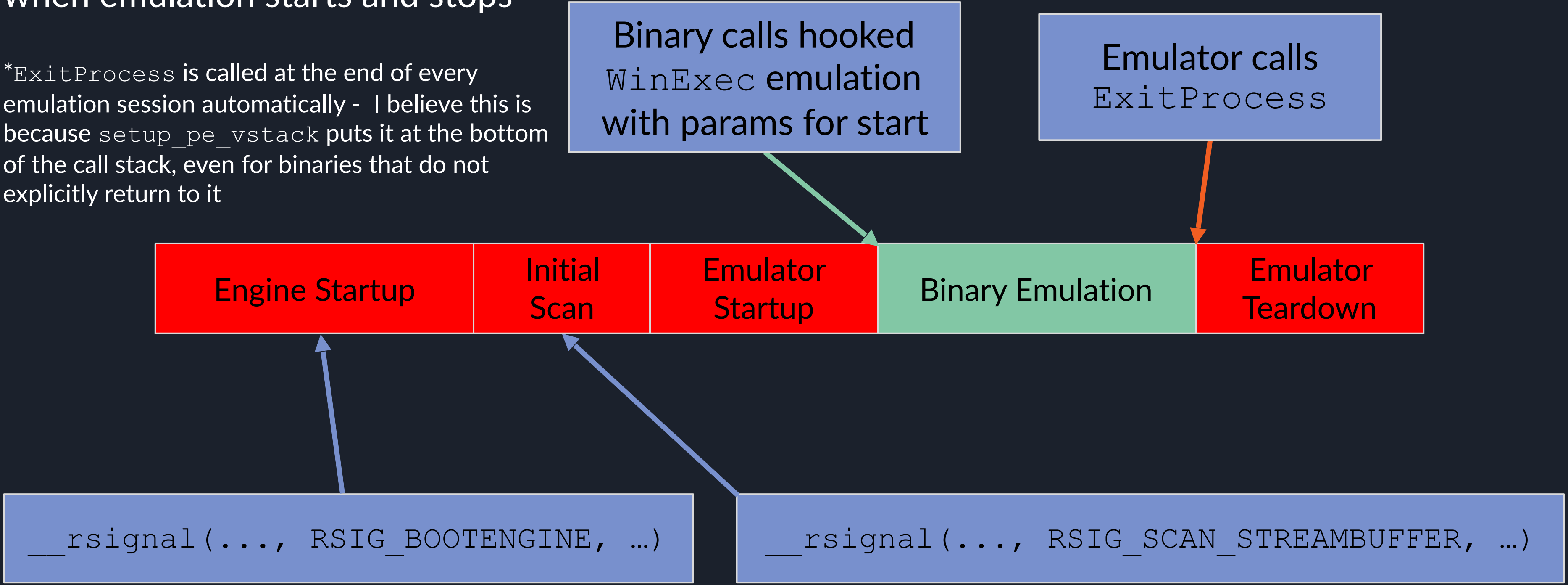
Tracing Timeline

Hooking Defender's emulation functions for `WinExec` and `ExitProcess` allows us to know when emulation starts and stops*

*`ExitProcess` is called at the end of every emulation session automatically - I believe this is because `setup_pe_vstack` puts it at the bottom of the call stack, even for binaries that do not explicitly return to it

Pintool must be enlightened about custom loaded `mpengine.dll` location - take callback stub ideas from Tavis Ormandy's `deepcover` Pintool

`github.com/taviso/loadlibrary/tree/master/coverage`



Binary calls hooked `WinExec` emulation with params for start

Emulator calls `ExitProcess`

Engine Startup

Initial Scan

Emulator Startup

Binary Emulation

Emulator Teardown

`__rsignal(..., RSIG_BOOTENGINE, …)`

`__rsignal(..., RSIG_SCAN_STREAMBUFFER, …)`

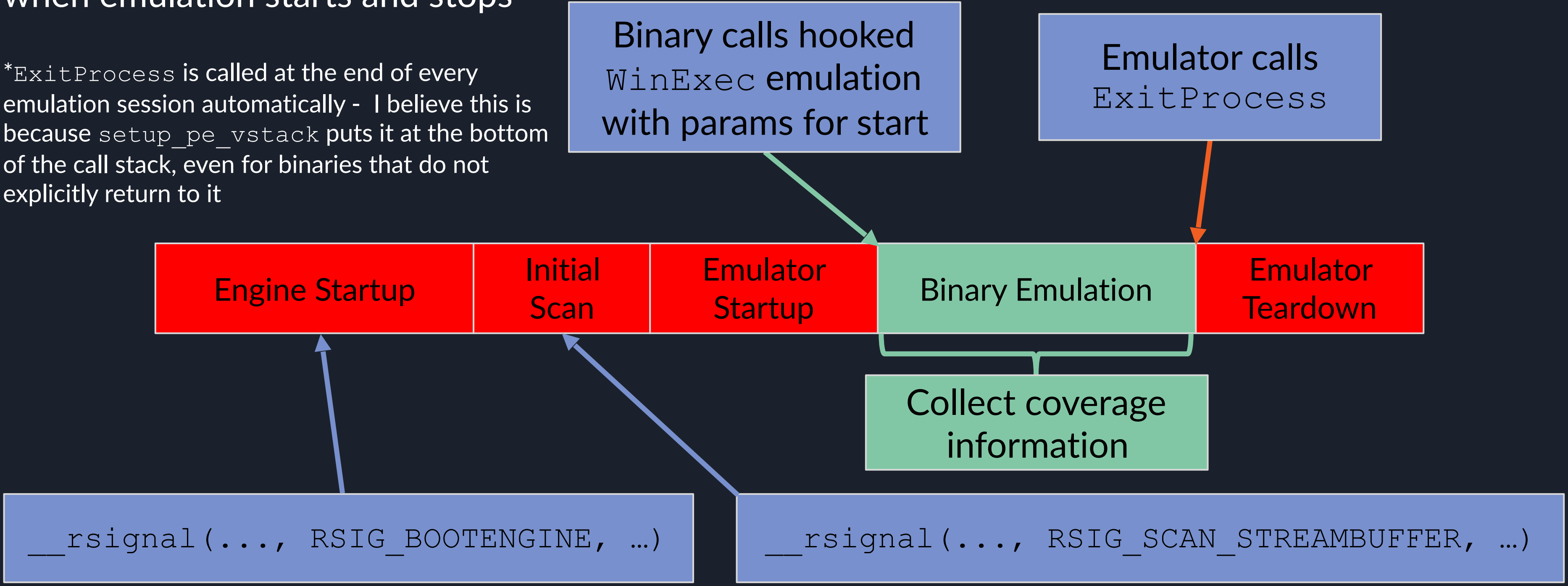
Tracing Timeline

Pintool must be enlightened about custom loaded `mpengine.dll` location - take callback stub ideas from Tavis Ormandy's `deepcover` Pintool

`github.com/taviso/loadlibrary/tree/master/coverage`

Hooking Defender's emulation functions for `WinExec` and `ExitProcess` allows us to know when emulation starts and stops*

*`ExitProcess` is called at the end of every emulation session automatically - I believe this is because `setup_pe_vstack` puts it at the bottom of the call stack, even for binaries that do not explicitly return to it



Binary calls hooked `WinExec` emulation with params for start

Emulator calls `ExitProcess`

Engine Startup

Initial Scan

Emulator Startup

Binary Emulation

Emulator Teardown

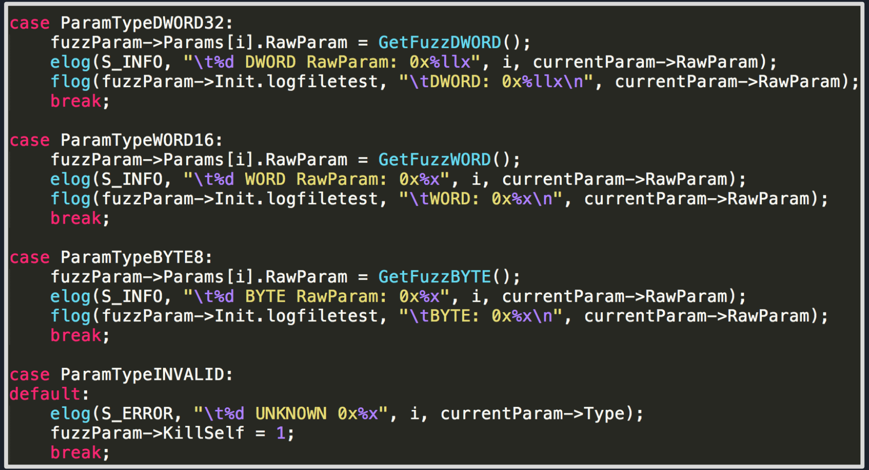
Collect coverage information

`__rsignal(..., RSIG_BOOTENGINE, …)`

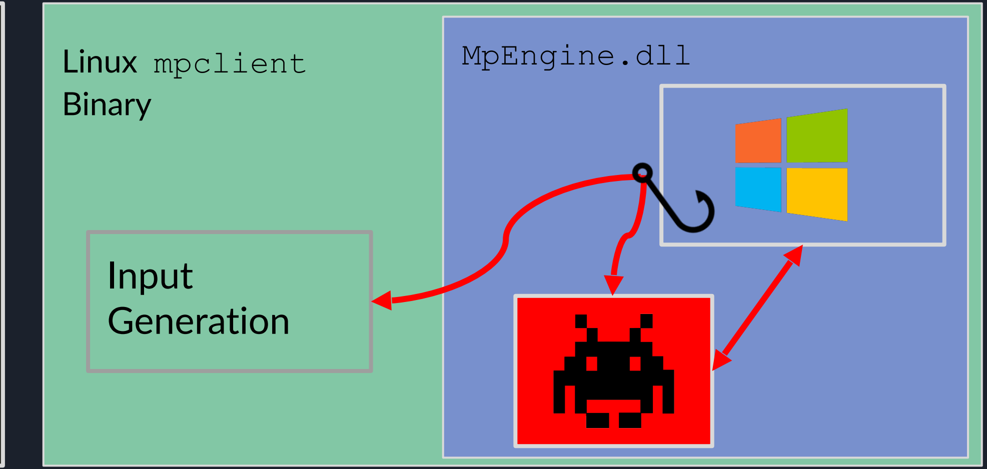
`__rsignal(..., RSIG_SCAN_STREAMBUFFER, …)`

Fuzzing Emulated APIs

- Create a binary that goes inside the emulator and repeatedly calls hooked `WinExec` function to request new data, then sends that data to functions with native emulations
- Buffers in memory passed to external hook function to populate with parameters
- Could do fuzzing in-emulator too, but this is easier for logging results



```
case ParamTypeDWORD32:
    fuzzParam->Params[i].RawParam = GetFuzzDWORD();
    elog(S_INFO, "\t%d DWORD RawParam: 0x%llx", i, currentParam->RawParam);
    flog(fuzzParam->Init.logfiletest, "\tDWORD: 0x%llx\n", currentParam->RawParam);
    break;

case ParamTypeWORD16:
    fuzzParam->Params[i].RawParam = GetFuzzWORD();
    elog(S_INFO, "\t%d WORD RawParam: 0x%x", i, currentParam->RawParam);
    flog(fuzzParam->Init.logfiletest, "\tWORD: 0x%x\n", currentParam->RawParam);
    break;

case ParamTypeBYTE8:
    fuzzParam->Params[i].RawParam = GetFuzzBYTE();
    elog(S_INFO, "\t%d BYTE RawParam: 0x%x", i, currentParam->RawParam);
    flog(fuzzParam->Init.logfiletest, "\tBYTE: 0x%x\n", currentParam->RawParam);
    break;

case ParamTypeINVALID:
default:
    elog(S_ERROR, "\t%d UNKNOWN 0x%x", i, currentParam->Type);
    fuzzParam->KillSelf = 1;
    break;
```



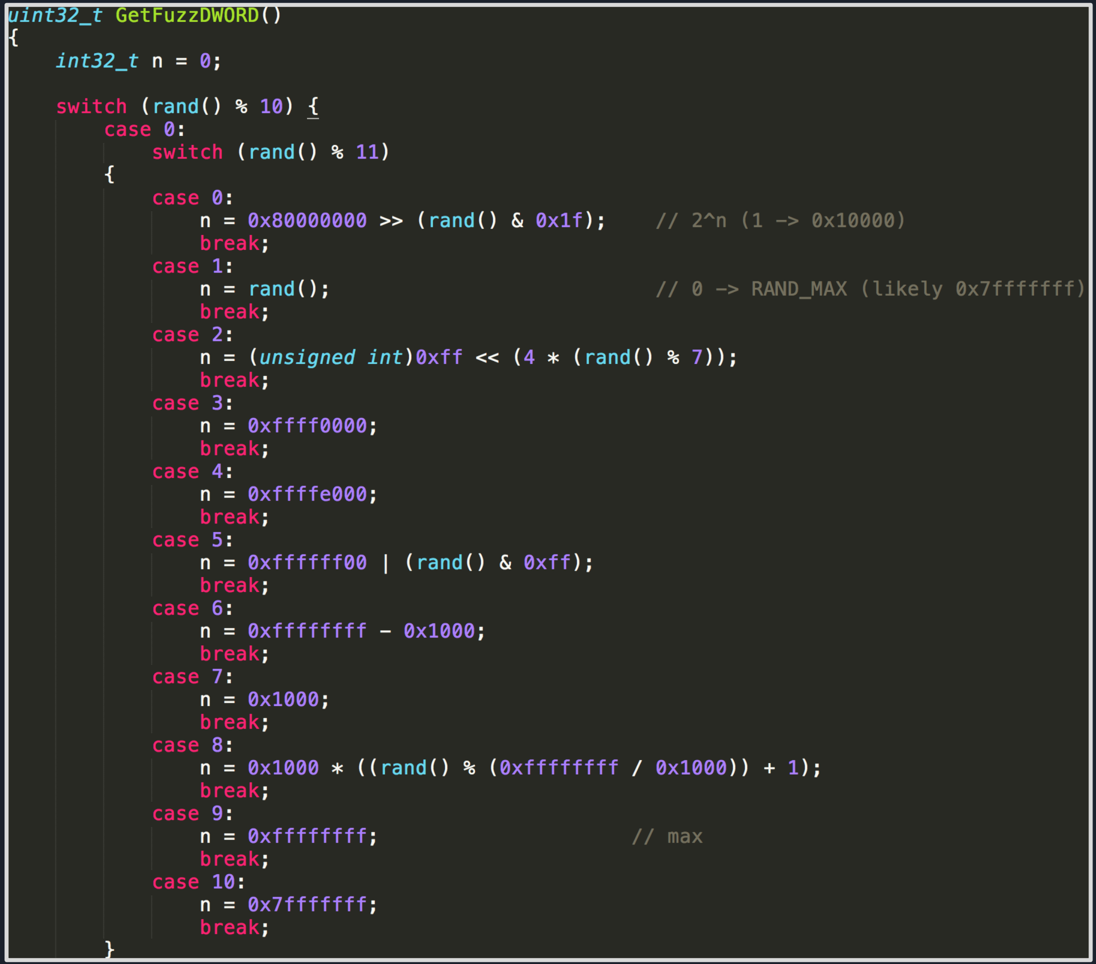
Input Generation

- Borrow OSX syscall fuzzer code from MWR Labs OSXFuzz project*

- Nothing fancy, just throw random values at native emulation handlers

- Re-seed `rand()` at the start of each emulation session, just save off seeds in a log

*github.com/mwrlabs/OSXFuzz



```
uint32_t GetFuzzDWORD()
{
    int32_t n = 0;

    switch (rand() % 10) {
        case 0:
            switch (rand() % 11)
        {
            case 0:
                n = 0x80000000 >> (rand() & 0x1f);    // 2^n (1 -> 0x10000)
                break;
            case 1:
                n = rand();                           // 0 -> RAND_MAX (likely 0x7fffffff)
                break;
            case 2:
                n = (unsigned int)0xff << (4 * (rand() % 7));
                break;
            case 3:
                n = 0xffff0000;
                break;
            case 4:
                n = 0xffffe000;
                break;
            case 5:
                n = 0xffffff00 | (rand() & 0xff);
                break;
            case 6:
                n = 0xffffffff - 0x1000;
                break;
            case 7:
                n = 0x1000;
                break;
            case 8:
                n = 0x1000 * ((rand() % (0xffffffff / 0x1000)) + 1);
                break;
            case 9:
                n = 0xffffffff;                     // max
                break;
            case 10:
                n = 0x7fffffff;
                break;
        }
```

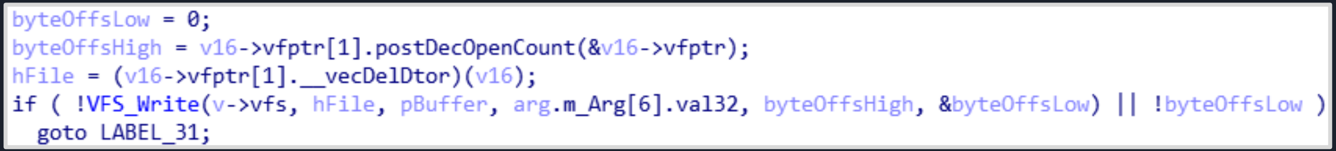
NtWriteFile Overflow

NtWriteFile is normally accessible and exported by ntdll.dll

- VFS_Write has to be triggered with special apicall

Tavis’ inputs get sanitized out by NtWriteFileWorker before it calls down to VFS_Write

```
LARGE_INTEGER L;
L.QuadPart =
0x2ff9ad29ffffc25;

NtWriteFile(
     hFile,
          NULL,
          NULL,
          NULL,
          &ioStatus,
          buf,
          0x1,
          &L,
          NULL);

L.QuadPart = 0x29548af5d7b3b7c;
NtWriteFile(
     hFile,
          NULL,
          NULL,
          NULL,
          &ioStatus,
          buf,
          0x1,
          &L,
          NULL);
```



```
byteOffsLow = 0;
byteOffsHigh = v16->vfptr[1].postDecOpenCount(&v16->vfptr);
hFile = (v16->vfptr[1].__vecDelDtor)(v16);
if ( !VFS_Write(v->vfs, hFile, pBuffer, arg.m_Arg[6].val32, byteOffsHigh, &byteOffsLow) || !byteOffsLow )
  goto LABEL_31;
```

NtWriteFile Overflow

NtWriteFile is normally accessible and exported by ntdll.dll

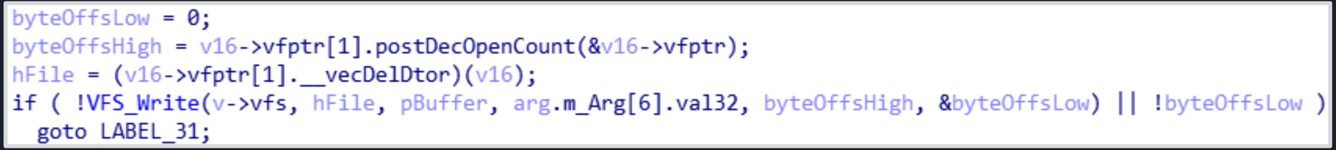
- VFS_Write has to be triggered with special apicall

Tavis' inputs get sanitized out by NtWriteFileWorker before it calls down to VFS_Write

I fuzzed NtWriteFile:

- ~7 minutes @ ~8,000 NtWriteFile calls / second
- Fuzzed Length arguments
- Reproduced Tavis' crash, alternate easier to reach code path through NtWriteFile

Unfortunately, patches for VFS_Write bug also fixed this



```
byteOffsLow = 0;
byteOffsHigh = v16->vfptr[1].postDecOpenCount(&v16->vfptr);
hFile = (v16->vfptr[1].__vecDelDtor)(v16);
if ( !VFS_Write(v->vfs, hFile, pBuffer, arg.m_Arg[6].val32, byteOffsHigh, &byteOffsLow) || !byteOffsLow )
  goto LABEL_31;
```

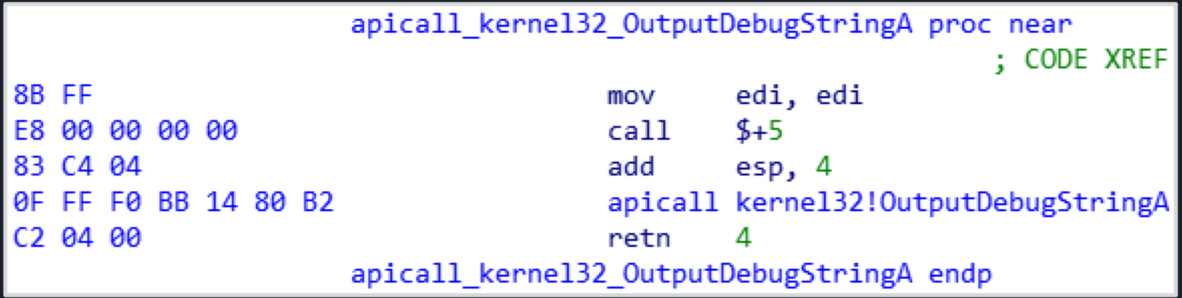
```
LARGE_INTEGER L;
L.QuadPart =
0x2ff9ad29fffffc25;

NtWriteFile(
     hFile,
          NULL,
          NULL,
          NULL,
          &ioStatus,
          buf,
          0x1,
          &L,
          NULL);

L.QuadPart = 0x29548af5d7b3b7c;
NtWriteFile(
     hFile,
          NULL,
          NULL,
          NULL,
          &ioStatus,
          buf,
          0x1,
          &L,
          NULL);
```

`apicall`

Custom “`apicall`” opcode used to trigger native emulation routines

`0F FF F0 [4 byte immediate]`

`apicall` instructions can be disassembled with an IDA Processor Extension Module



```
                  apicall_kernel32_OutputDebugStringA proc near
                                                           ; CODE XREF
8B FF                                 mov     edi, edi
E8 00 00 00 00                        call    $+5
83 C4 04                              add     esp, 4
0F FF F0 BB 14 80 B2                  apicall kernel32!OutputDebugStringA
C2 04 00                              retn    4
                  apicall_kernel32_OutputDebugStringA endp
```

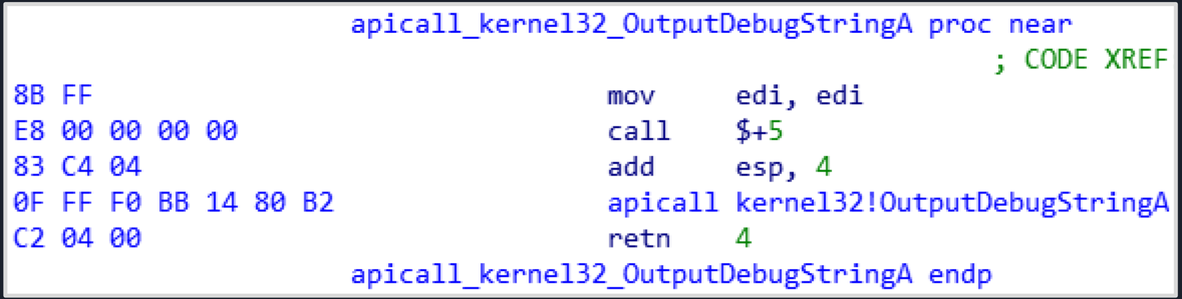
apicall

Custom “apicall” opcode used to trigger native emulation routines

```
0F FF F0 [4 byte immediate]
```

```
immediate = crc32(DLL name, all caps) ^ crc32(function name)
```

apicall instructions can be disassembled with an IDA Processor Extension Module



```
                      apicall_kernel32_OutputDebugStringA proc near
                                                             ; CODE XREF
8B FF                                     mov     edi, edi
E8 00 00 00 00                            call    $+5
83 C4 04                                  add     esp, 4
0F FF F0 BB 14 80 B2                      apicall kernel32!OutputDebugStringA
C2 04 00                                  retn    4
                      apicall_kernel32_OutputDebugStringA endp
```

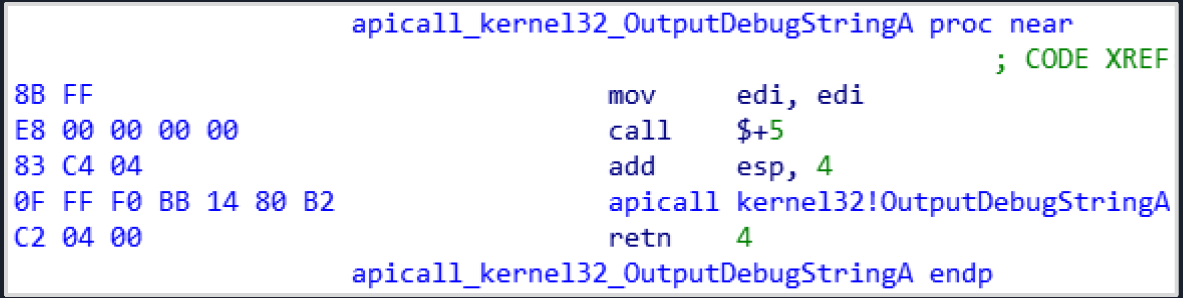
apicall

Custom "apicall" opcode used to trigger native emulation routines

```
0F FF F0 [4 byte immediate]
```

```
immediate = crc32(DLL name, all caps) ^ crc32(function name)

0xB28014BB = crc32("KERNEL32.DLL") ^ crc32("OutputDebugStringA")
```

apicall instructions can be disassembled with an IDA Processor Extension Module



```
                    apicall_kernel32_OutputDebugStringA proc near
                                                               ; CODE XREF
8B FF                                  mov      edi, edi
E8 00 00 00 00                         call     $+5
83 C4 04                               add      esp, 4
0F FF F0 BB 14 80 B2                   apicall  kernel32!OutputDebugStringA
C2 04 00                               retn     4
                    apicall_kernel32_OutputDebugStringA endp
```

apicall

Custom “apicall” opcode used to trigger native emulation routines

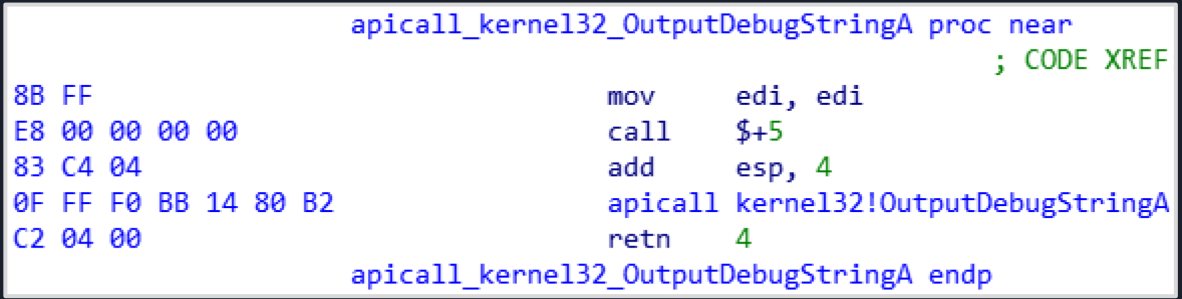
```
0F FF F0 [4 byte immediate]
```

```
immediate = crc32(DLL name, all caps) ^ crc32(function name)
```

```
0xB28014BB = crc32(“KERNEL32.DLL”) ^ crc32(“OutputDebugStringA”)
```

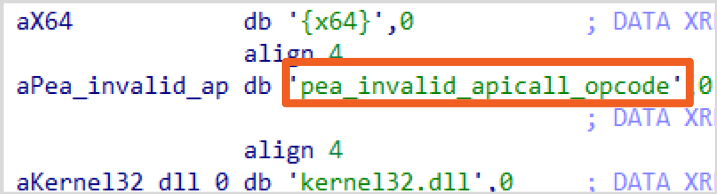
```
0F FF F0 BB 14 80 B2
apicall kernel32!OutPutDebugStringA
```

apicall instructions can be disassembled with an IDA Processor Extension Module



```
                    apicall_kernel32_OutputDebugStringA proc near
                                                           ; CODE XREF
8B FF                                     mov      edi, edi
E8 00 00 00 00                            call     $+5
83 C4 04                                  add      esp, 4
0F FF F0 BB 14 80 B2                      apicall  kernel32!OutputDebugStringA
C2 04 00                                  retn     4
                    apicall_kernel32_OutputDebugStringA endp
```

Locking Down apicall

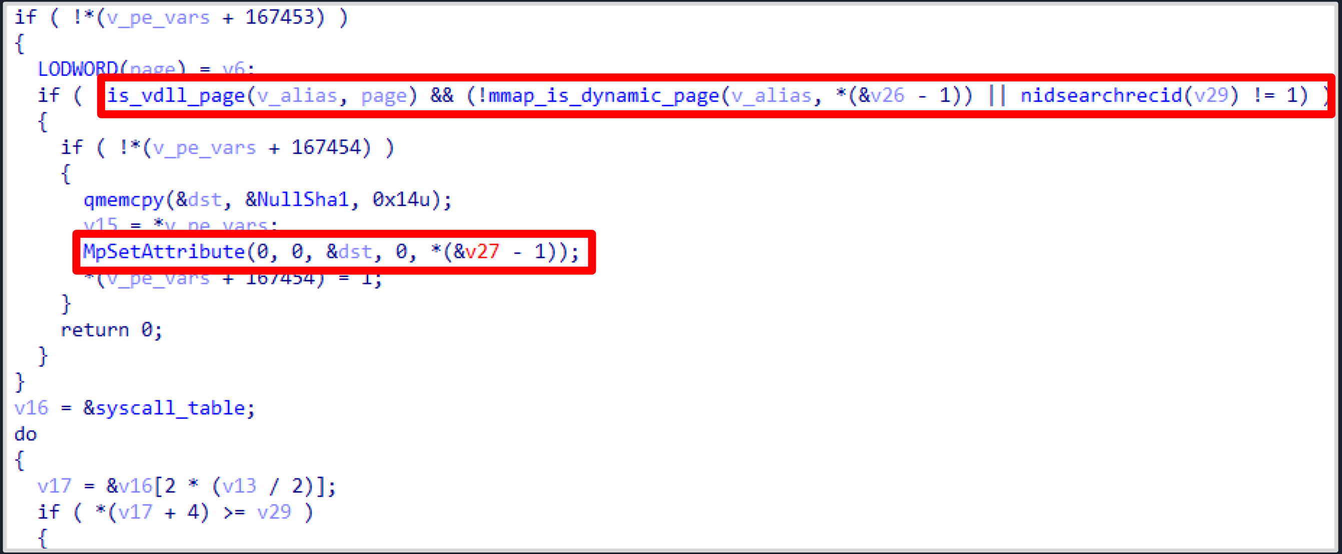


```
aX64            db '{x64}',0                    ; DATA XR
                align 4
aPea_invalid_ap db 'pea_invalid_apicall_opcode',0
                                                ; DATA XR
                align 4
aKernel32_dll_0 db 'kernel32.dll',0             ; DATA XR
```

New AV heuristic trait added

is_vdll_page call added to __call_api_by_crc in 6/20/2017 mpengine.dll build - is the apicall instruction coming from a VDLL?

Can't just trigger apicall from malware .text section or otherwise malware-created memory (eg: rwx allocation) anymore



```
if ( !*(v_pe_vars + 167453) )
{
  LODWORD(page) = v6;
  if ( is_vdll_page(v_alias, page) && (!mmap_is_dynamic_page(v_alias, *(&v26 - 1)) || nidsearchrecid(v29) != 1) )
  {
    if ( !*(v_pe_vars + 167454) )
    {
      qmemcpy(&dst, &NullSha1, 0x14u);
      v15 = *v_pe_vars;
      MpSetAttribute(0, 0, &dst, 0, *(&v27 - 1));
      *(v_pe_vars + 167454) = 1;
    }
    return 0;
  }
}
v16 = &syscall_table;
do
{
  v17 = &v16[2 * (v13 / 2)];
  if ( *(v17 + 4) >= v29 )
  {
```

If apicall did not come from a VDLL, set a heuristic and deny it

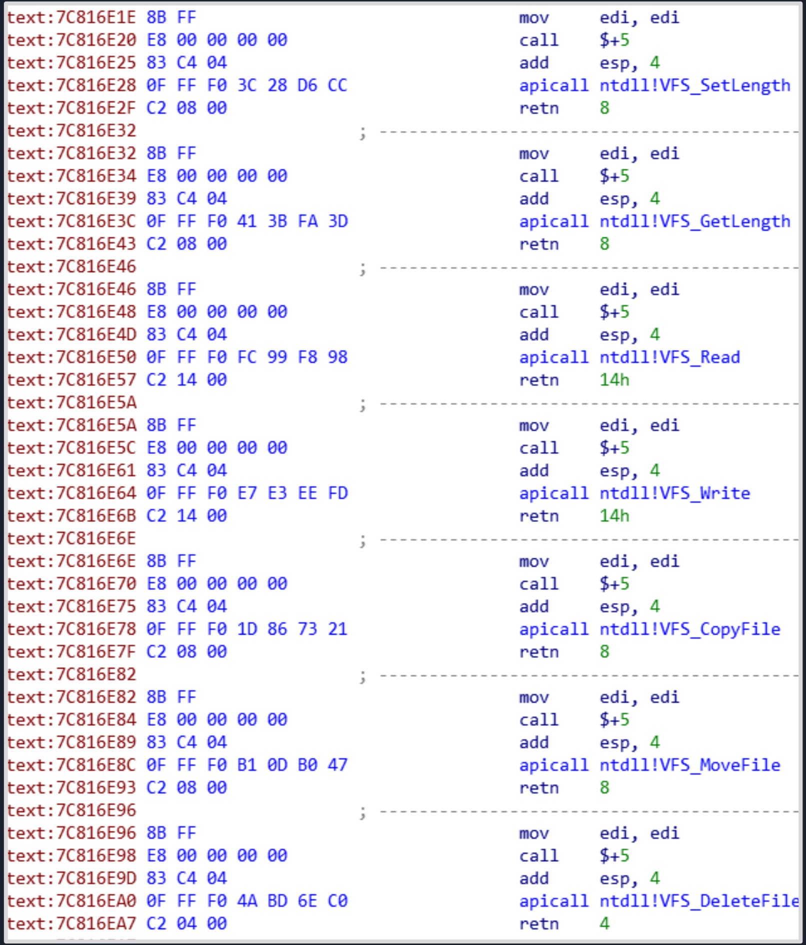
Proceed with processing if apicall is ok

Bypass

- `apicall` stubs are located throughout VDLLs
- They can be located in memory and called directly by malware, with attacker controlled arguments
    - Passes `is_vdll_page` checks

Response from MSFT: “We did indeed make some changes to make this interface harder to reach from the code we’re emulating -however, that was never intended to be a trust boundary.

Accessing the internal APIs exposed to the emulation code is not a security vulnerability...”



```
text:7C816E1E 8B FF                          mov     edi, edi
text:7C816E20 E8 00 00 00 00                 call    $+5
text:7C816E25 83 C4 04                       add     esp, 4
text:7C816E28 0F FF F0 3C 28 D6 CC           apicall ntdll!VFS_SetLength
text:7C816E2F C2 08 00                       retn    8
text:7C816E32                    ; ---------------------------------------------
text:7C816E32 8B FF                          mov     edi, edi
text:7C816E34 E8 00 00 00 00                 call    $+5
text:7C816E39 83 C4 04                       add     esp, 4
text:7C816E3C 0F FF F0 41 3B FA 3D           apicall ntdll!VFS_GetLength
text:7C816E43 C2 08 00                       retn    8
text:7C816E46                    ; ---------------------------------------------
text:7C816E46 8B FF                          mov     edi, edi
text:7C816E48 E8 00 00 00 00                 call    $+5
text:7C816E4D 83 C4 04                       add     esp, 4
text:7C816E50 0F FF F0 FC 99 F8 98           apicall ntdll!VFS_Read
text:7C816E57 C2 14 00                       retn    14h
text:7C816E5A                    ; ---------------------------------------------
text:7C816E5A 8B FF                          mov     edi, edi
text:7C816E5C E8 00 00 00 00                 call    $+5
text:7C816E61 83 C4 04                       add     esp, 4
text:7C816E64 0F FF F0 E7 E3 EE FD           apicall ntdll!VFS_Write
text:7C816E6B C2 14 00                       retn    14h
text:7C816E6E                    ; ---------------------------------------------
text:7C816E6E 8B FF                          mov     edi, edi
text:7C816E70 E8 00 00 00 00                 call    $+5
text:7C816E75 83 C4 04                       add     esp, 4
text:7C816E78 0F FF F0 1D 86 73 21           apicall ntdll!VFS_CopyFile
text:7C816E7F C2 08 00                       retn    8
text:7C816E82                    ; ---------------------------------------------
text:7C816E82 8B FF                          mov     edi, edi
text:7C816E84 E8 00 00 00 00                 call    $+5
text:7C816E89 83 C4 04                       add     esp, 4
text:7C816E8C 0F FF F0 B1 0D B0 47           apicall ntdll!VFS_MoveFile
text:7C816E93 C2 08 00                       retn    8
text:7C816E96                    ; ---------------------------------------------
text:7C816E96 8B FF                          mov     edi, edi
text:7C816E98 E8 00 00 00 00                 call    $+5
text:7C816E9D 83 C4 04                       add     esp, 4
text:7C816EA0 0F FF F0 4A BD 6E C0           apicall ntdll!VFS_DeleteFile
text:7C816EA7 C2 04 00                       retn    4
```

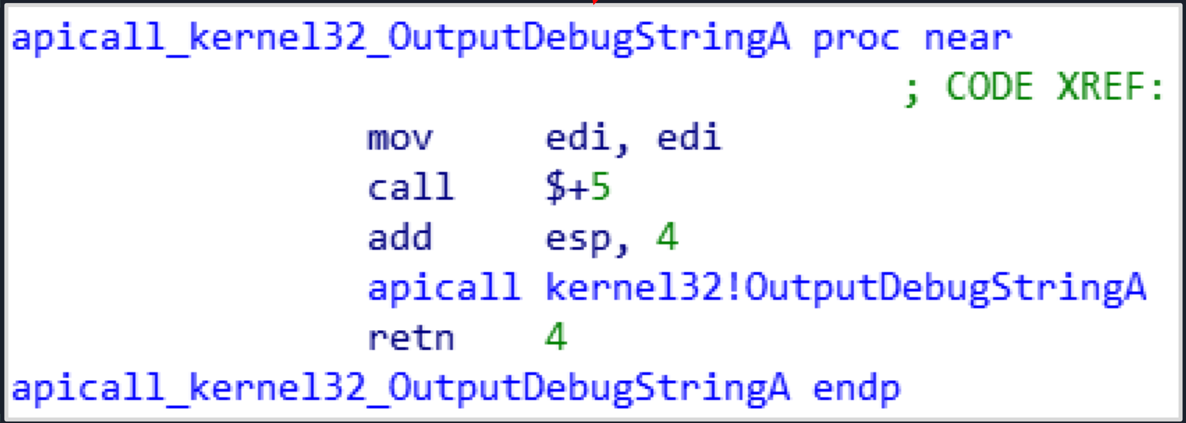
Bypass Example

OutputDebugStringA can be normally hit from kernel32, so this is ultimately just a unique way of doing that

```
VOID OutputDebugStringA_APICALL(PCHAR msg)
{
    typedef VOID(*PODS)(PCHAR);
    HMODULE k32base = LoadLibraryA(“kernel32.dll”);
    PODS apicallODS = (PODS)((PBYTE)k32base + 0x16d4e);
    apicallODS(msg);
}
```

Kernel32 base offset: 0x16d4e

Comes from kernel32 VDLL, so passes is_vdll_page checks



```
apicall_kernel32_OutputDebugStringA proc near
                                        ; CODE XREF:
                mov     edi, edi
                call    $+5
                add     esp, 4
                apicall kernel32!OutputDebugStringA
                retn    4
apicall_kernel32_OutputDebugStringA endp
```

Outline

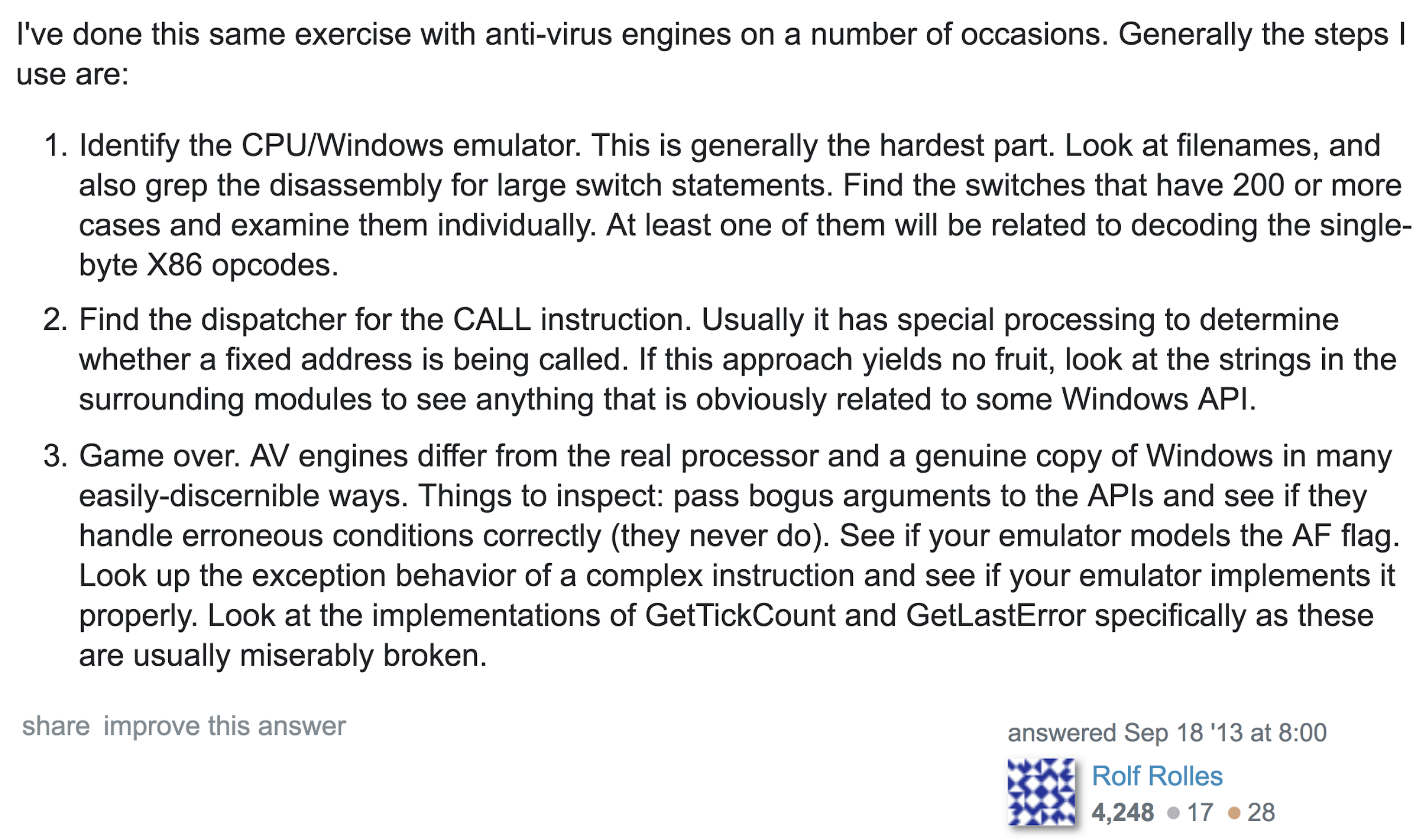
1. Introduction
2. Tooling & Process
3. Discussion
4. Conclusion

Reverse Engineer Intuitions

- It’s easy to detect for emulator (or file format unpacker) presence - test an `EICAR` dropper
- Everyone has to emulate `Sleep()` with custom code
- Everyone emulates `cpuid`
- Everyone emulates `rstsc`, but messes up `rdtscp`
- Emulators have lots of strings - these can be found in memory dumps to help identify emulator code

- Everyone builds custom tools when doing offensive research, but this is especially true for AV RE

Reverse Engineer Intuitions - Rolf Rolles in 2013



I've done this same exercise with anti-virus engines on a number of occasions. Generally the steps I use are:

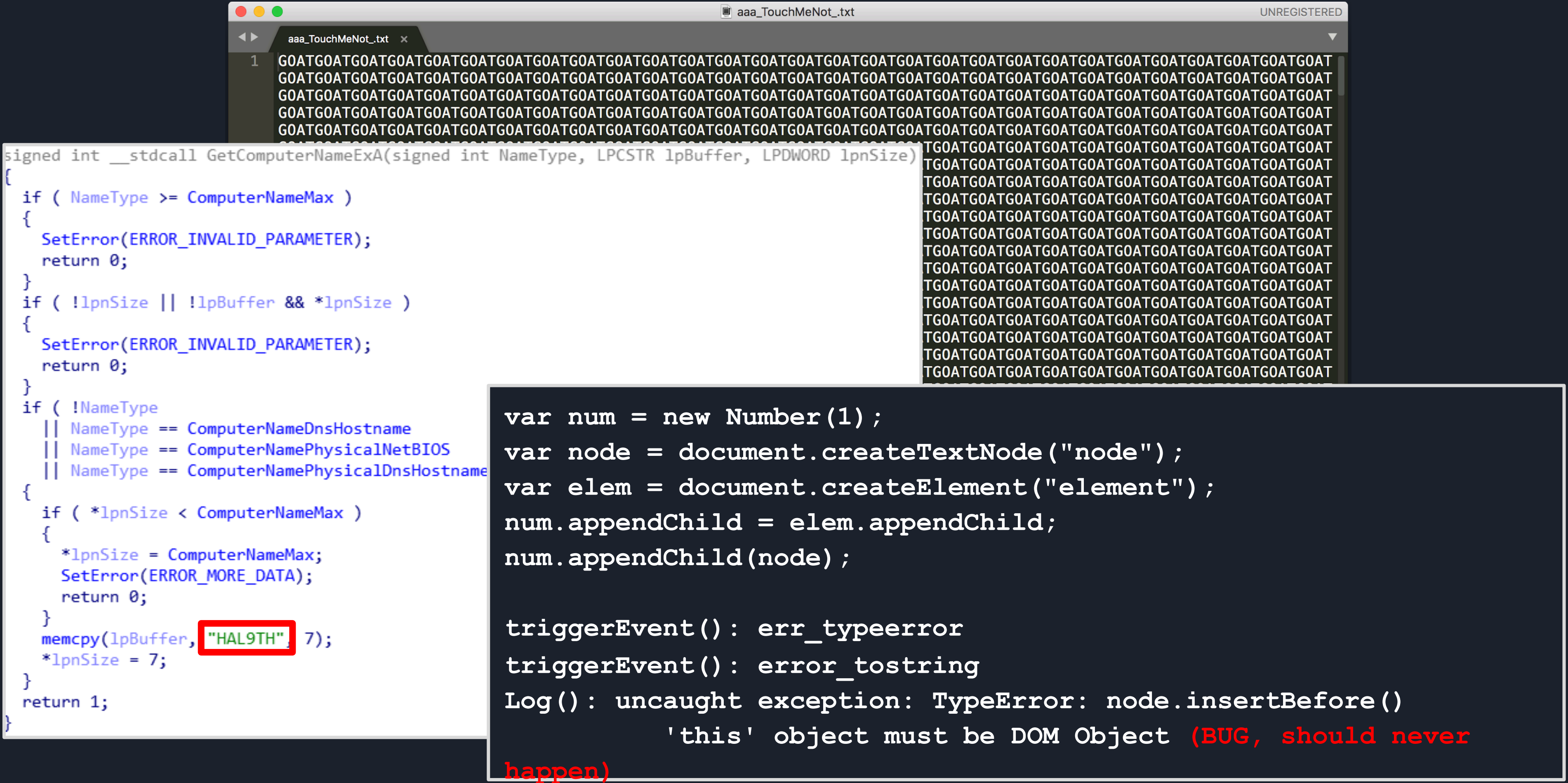
1. Identify the CPU/Windows emulator. This is generally the hardest part. Look at filenames, and also grep the disassembly for large switch statements. Find the switches that have 200 or more cases and examine them individually. At least one of them will be related to decoding the single-byte X86 opcodes.
2. Find the dispatcher for the CALL instruction. Usually it has special processing to determine whether a fixed address is being called. If this approach yields no fruit, look at the strings in the surrounding modules to see anything that is obviously related to some Windows API.
3. Game over. AV engines differ from the real processor and a genuine copy of Windows in many easily-discernible ways. Things to inspect: pass bogus arguments to the APIs and see if they handle erroneous conditions correctly (they never do). See if your emulator models the AF flag. Look up the exception behavior of a complex instruction and see if your emulator implements it properly. Look at the implementations of GetTickCount and GetLastError specifically as these are usually miserably broken.

answered Sep 18 '13 at 8:00

Rolf Rolles

4,248 ● 17 ● 28

Programmer “Easter Eggs”



```
GOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOAT
GOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOAT
GOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOAT
GOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOAT
GOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOAT
TGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOAT
TGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOAT
TGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOAT
TGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOAT
TGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOAT
TGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOAT
TGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOAT
TGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOAT
TGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOAT
TGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOAT
TGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOAT
TGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOAT
TGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOAT
TGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOATGOAT
```

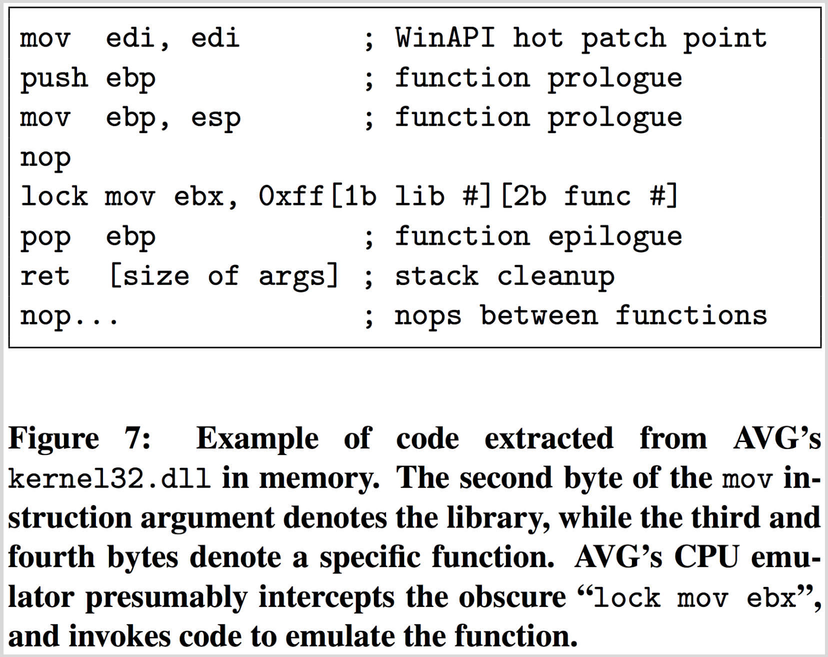
```
signed int __stdcall GetComputerNameExA(signed int NameType, LPCSTR lpBuffer, LPDWORD lpnSize)
{
  if ( NameType >= ComputerNameMax )
  {
    SetError(ERROR_INVALID_PARAMETER);
    return 0;
  }
  if ( !lpnSize || !lpBuffer && *lpnSize )
  {
    SetError(ERROR_INVALID_PARAMETER);
    return 0;
  }
  if ( !NameType
    || NameType == ComputerNameDnsHostname
    || NameType == ComputerNamePhysicalNetBIOS
    || NameType == ComputerNamePhysicalDnsHostname
  {
    if ( *lpnSize < ComputerNameMax )
    {
      *lpnSize = ComputerNameMax;
      SetError(ERROR_MORE_DATA);
      return 0;
    }
    memcpy(lpBuffer, "HAL9TH", 7);
    *lpnSize = 7;
  }
  return 1;
}
```

```
var num = new Number(1);
var node = document.createTextNode("node");
var elem = document.createElement("element");
num.appendChild = elem.appendChild;
num.appendChild(node);

triggerEvent(): err_typeerror
triggerEvent(): error_tostring
Log(): uncaught exception: TypeError: node.insertBefore()
          'this' object must be DOM Object (BUG, should never
happen)
```

In-Emulator Signaling

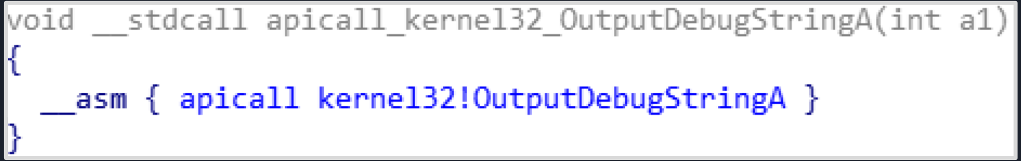
Attackers can discover in-emulator control operations

Why not just use `int/syscall`?

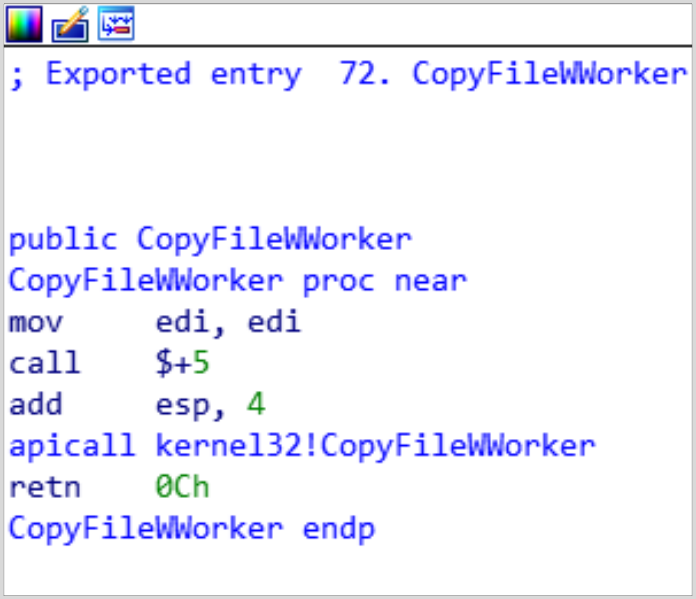


```
mov  edi, edi          ; WinAPI hot patch point
push ebp               ; function prologue
mov  ebp, esp          ; function prologue
nop
lock mov ebx, 0xff[1b lib #][2b func #]
pop  ebp               ; function epilogue
ret  [size of args]    ; stack cleanup
nop...                 ; nops between functions
```

Figure 7: Example of code extracted from AVG's `kernel32.dll` in memory. The second byte of the `mov` instruction argument denotes the library, while the third and fourth bytes denote a specific function. AVG's CPU emulator presumably intercepts the obscure "`lock mov ebx`", and invokes code to emulate the function.



```
void __stdcall apicall_kernel32_OutputDebugStringA(int a1)
{
  __asm { apicall kernel32!OutputDebugStringA }
}
```

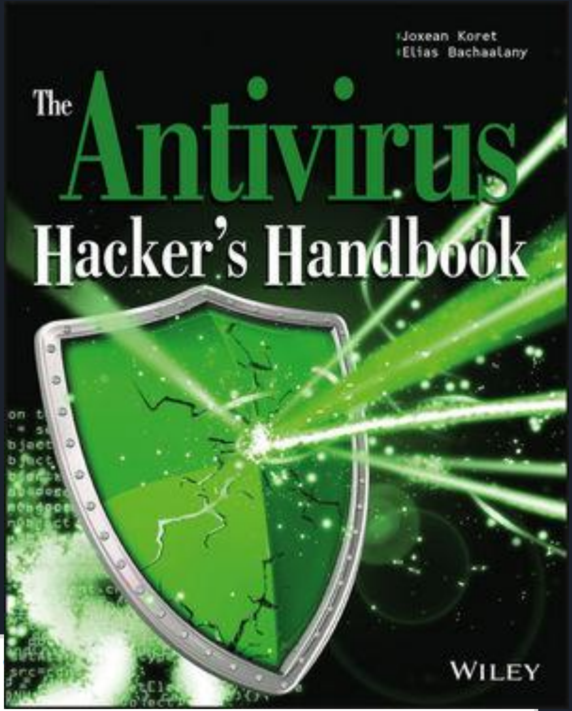


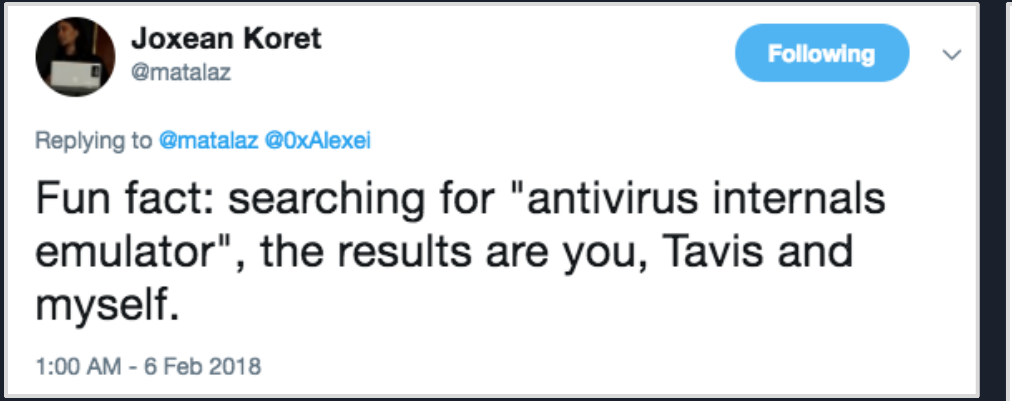
```
; Exported entry  72. CopyFileWWorker



public CopyFileWWorker
CopyFileWWorker proc near
mov     edi, edi
call    $+5
add     esp, 4
apicall kernel32!CopyFileWWorker
retn    0Ch
CopyFileWWorker endp
```

Antivirus Reverse Engineering

- People constantly talk about what AVs can or can't do, and how/where they are vulnerable
- These claims are mostly backed up by Tavis Ormandy's work at Project Zero and a handful of other conference talks, papers, and blog posts

- I hope we'll see more AV research in the future

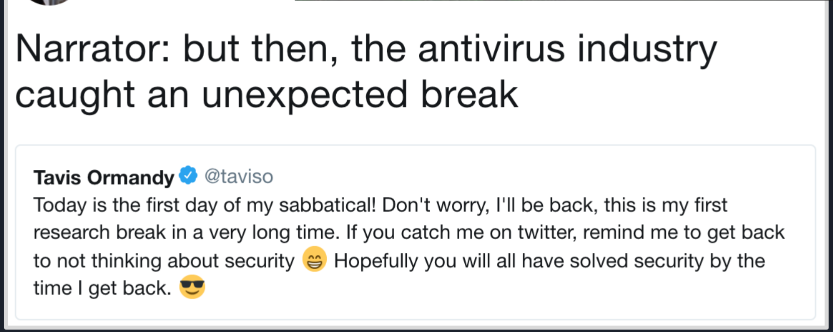



Joxean Koret
@matalaz

Replying to @matalaz @0xAlexei

Fun fact: searching for "antivirus internals emulator", the results are you, Tavis and myself.

1:00 AM - 6 Feb 2018


Stefano Zanero
@raistolo

Narrator: but then, the antivirus industry caught an unexpected break

Tavis Ormandy @taviso

Today is the first day of my sabbatical! Don't worry, I'll be back, this is my first research break in a very long time. If you catch me on twitter, remind me to get back to not thinking about security 😁 Hopefully you will all have solved security by the time I get back. 😎

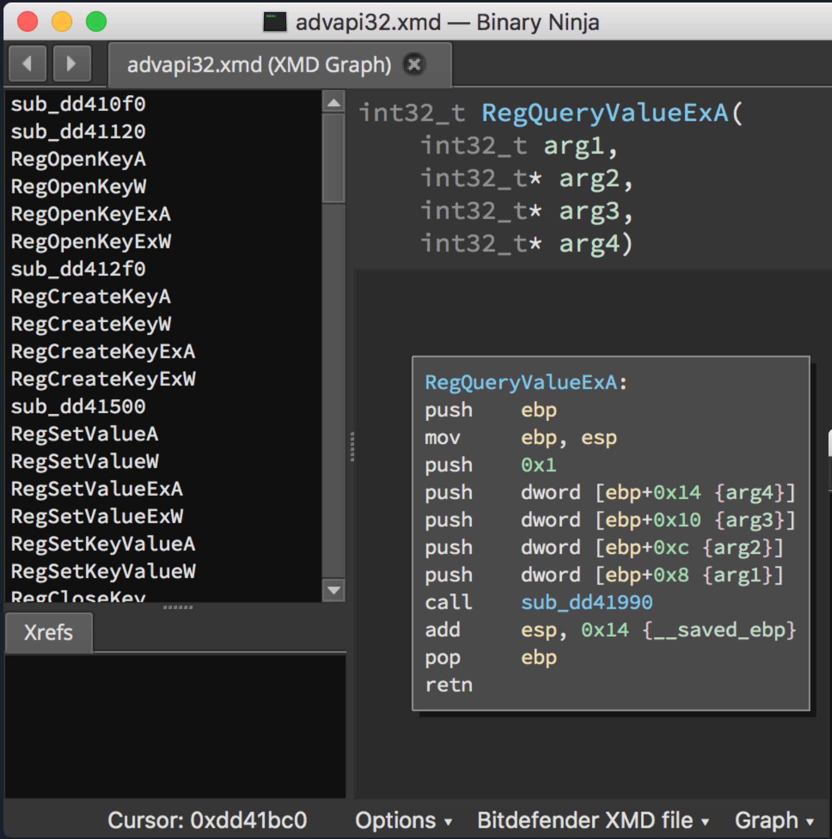
Security Through Obscurity?

- Preventing reverse engineering is futile
  - Obfuscation and custom binary formats don’t stop RE, and can be overcome with one-time effort
  - Side channel analyses like “AVLeak” are also possible

- Introspectibility and debugability are poor → only *motivated competent* adversaries will perform RE
  - Malicious actors *already are* - search any unique string from my presentations - you’ll find malware samples from long before I presented

Custom Binary Format Example: Bitdefender XMDs

Custom Binary Ninja loader:
~150 LoC, 4 hours of work



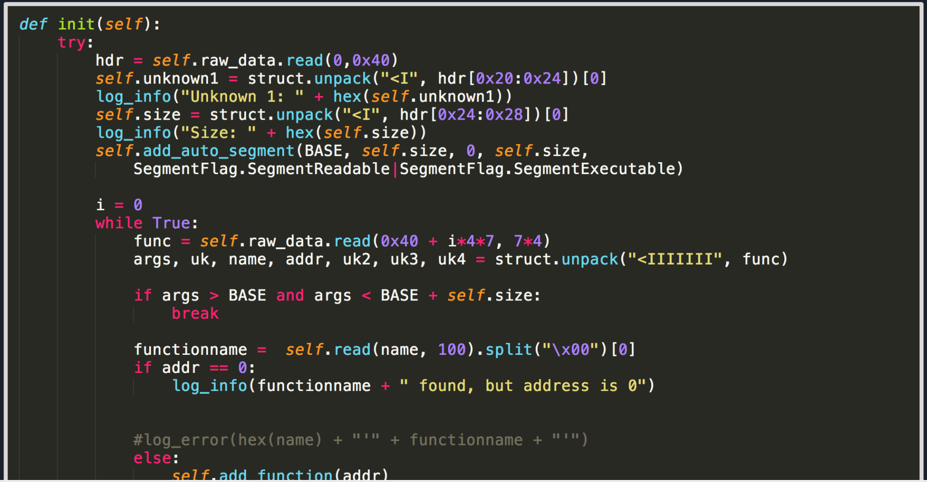
advapi32.xmd — Binary Ninja

advapi32.xmd (XMD Graph)

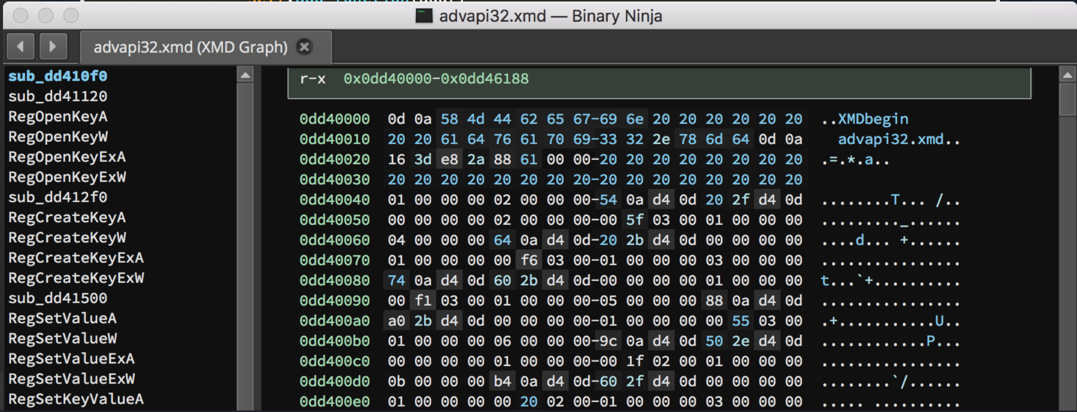
```
sub_dd410f0
sub_dd41120
RegOpenKeyA
RegOpenKeyW
RegOpenKeyExA
RegOpenKeyExW
sub_dd412f0
RegCreateKeyA
RegCreateKeyW
RegCreateKeyExA
RegCreateKeyExW
sub_dd41500
RegSetValueA
RegSetValueW
RegSetValueExA
RegSetValueExW
RegSetKeyValueA
RegSetKeyValueW
RegCloseKey
```

```
int32_t RegQueryValueExA(
    int32_t arg1,
    int32_t* arg2,
    int32_t* arg3,
    int32_t* arg4)
```

```
RegQueryValueExA:
push    ebp
mov     ebp, esp
push    0x1
push    dword [ebp+0x14 {arg4}]
push    dword [ebp+0x10 {arg3}]
push    dword [ebp+0xc {arg2}]
push    dword [ebp+0x8 {arg1}]
call    sub_dd41990
add     esp, 0x14 {__saved_ebp}
pop     ebp
retn
```



```python
def init(self):
    try:
        hdr = self.raw_data.read(0,0x40)
        self.unknown1 = struct.unpack("<I", hdr[0x20:0x24])[0]
        log_info("Unknown 1: " + hex(self.unknown1))
        self.size = struct.unpack("<I", hdr[0x24:0x28])[0]
        log_info("Size: " + hex(self.size))
        self.add_auto_segment(BASE, self.size, 0, self.size,
            SegmentFlag.SegmentReadable|SegmentFlag.SegmentExecutable)

        i = 0
        while True:
            func = self.raw_data.read(0x40 + i*4*7, 7*4)
            args, uk, name, addr, uk2, uk3, uk4 = struct.unpack("<IIIIIII", func)

            if args > BASE and args < BASE + self.size:
                break

            functionname =  self.read(name, 100).split("\x00")[0]
            if addr == 0:
                log_info(functionname + " found, but address is 0")


            #log_error(hex(name) + "'" + functionname + "'")
            else:
                self.add_function(addr)
```



advapi32.xmd — Binary Ninja

advapi32.xmd (XMD Graph)

```
sub_dd410f0
sub_dd41120
RegOpenKeyA
RegOpenKeyW
RegOpenKeyExA
RegOpenKeyExW
sub_dd412f0
RegCreateKeyA
RegCreateKeyW
RegCreateKeyExA
RegCreateKeyExW
sub_dd41500
RegSetValueA
RegSetValueW
RegSetValueExA
RegSetValueExW
RegSetKeyValueA
```

```
r-x  0x0dd40000-0x0dd46188

0dd40000  0d 0a 58 4d 44 62 65 67-69 6e 20 20 20 20 20 20  ..XMDbegin
0dd40010  20 20 61 64 76 61 70 69-33 32 2e 78 6d 64 0d 0a    advapi32.xmd..
0dd40020  16 3d e8 2a 88 61 00 00-20 20 20 20 20 20 20 20  .=.*.a..
0dd40030  20 20 20 20 20 20 20 20-20 20 20 20 20 20 20 20
0dd40040  01 00 00 00 02 00 00 00-54 0a d4 0d 20 2f d4 0d  ........T... /..
0dd40050  00 00 00 00 02 00 00 00-00 5f 03 00 01 00 00 00  ........._......
0dd40060  04 00 00 00 64 0a d4 0d-20 2b d4 0d 00 00 00 00  ....d... +......
0dd40070  01 00 00 00 00 f6 03 00-01 00 00 00 03 00 00 00  ................
0dd40080  74 0a d4 0d 60 2b d4 0d-00 00 00 00 01 00 00 00  t...`+..........
0dd40090  00 f1 03 00 01 00 00 00-05 00 00 00 88 0a d4 0d  ................
0dd400a0  a0 2b d4 0d 00 00 00 00-01 00 00 00 00 55 03 00  .+..........U..
0dd400b0  01 00 00 00 06 00 00 00-9c 0a d4 0d 50 2e d4 0d  ............P...
0dd400c0  00 00 00 00 01 00 00 00-00 1f 02 00 01 00 00 00  ................
0dd400d0  0b 00 00 00 b4 0a d4 0d-60 2f d4 0d 00 00 00 00  ........`/......
0dd400e0  01 00 00 00 00 20 02 00-01 00 00 00 03 00 00 00  ..... ..........
```

Emulator Exploitation

- Emulators, like web browsers, provide the primitives necessary for modern binary exploitation

- Micro-level: Software attack surface is immense, and the software runs at high privilege on the OS

- Macro-level: For IT organizations, AV software is similar - high privilege within a network, and adds attack surface to your most sensitive assets

- AV engines *seem* intuitively very easy to sandbox

Outline

1. Introduction
2. Tooling & Process
3. Discussion
4. Conclusion

Code & More Information

`github.com/0xAlexei`

Code release:

- `OutputDebugStringA` hooking
- "Malware" binary to go inside the emulator
- Some IDA scripts, including `apicall` disassembler

Article in PoC||GTFO 0x19:

- `OutputDebugStringA` hooking
- Patch diffing and `apicall` bypass
- `apicall` disassembly with IDA processor extension module

Conclusion

@0xAlexei

Open DMs

1. I had a great time reverse engineering Windows Defender - seriously cool software
2. REs will create custom tools to address AV complexity
3. Resistance to RE is futile, so be smart about design

Thank You:

- Tavis Ormandy & Natalie Silvanovich @ Google P0 - exposing the engine, `mpclient`, sharing ideas
- Mark - hooking ideas
- Joxean Koret - OG AV hacker
- Virus Bulletin - hosting me and editing my paper

JS Engine & Emulator slides:

bit.ly/2qio857

bit.ly/2CxyZ31

github.com/0xAlexei

Turn on virus protection

Virus protection is turned off. Tap or click to turn on Windows Defender.