Anti-Analysis

An overview of common techniques

Giovanni Lagorio

giovanni.lagorio@unige.it
https://csec.it/people/giovanni_lagorio
Twitter & GitHub: zxgio

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
University of Genova, Italy

www.zenhack.it

Outline

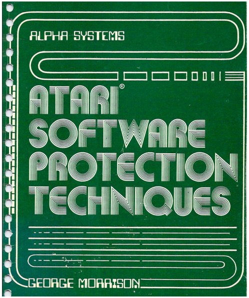
Introduction

- Some programs don't like being analyzed
  - E.g., many malware samples, videogames, ...
- They try to detect/disrupt an analysis environment
- Different techniques aim at different targets
  - packing/code obfuscation $\rightarrow$ disassembly/static-analysis
  - anti-debugging $\rightarrow$ debuggers/dynamic-analysis
  - anti-VM/sandbox $\rightarrow$ dynamic-analysis
  - ...

A lot of techniques, we'll discuss general ideas and some examples

A quote from 1983

Can any program be made totally uncopyable? This question gets a qualified no. For most practical purposes, any software can be pirated. No matter how complex the protection technique, there are people who can break it. Any protection technique invented by man can be broken by man



Outline

Anti-disassembly: multiple paths (1/2)

Instruction	Encoding
NOP	66 90
NOP DWORD PTR [EAX]	0F 1F 00
NOP DWORD PTR [EAX+00]	0F 1F 40 00
NOP DWORD PTR [EAX+EAX + 00]	0F 1F 44 00 00
NOP WORD PTR [EAX+EAX + 00]	66 0F 1F 44 00 00
NOP DWORD PTR [EAX+00000000]	0F 1F 80 00 00 00 00
NOP DWORD PTR [EAX+EAX + 00000000]	0F 1F 84 00 00 00 00 00
NOP WORD PTR [EAX+EAX + 00000000]	66 0F 1F 84 00 00 00 00 00

For the nine-byte `NOP`, the first byte (`66`) is an instruction prefix for overriding the operand-size. The following two bytes (`0F 1F`) is the opcode. The fourth byte (`84`) is the so called Mod R/M byte, which essentially describes the format of the operand. The last five bytes (`00 00 00 00 00`) describe the memory operand. Note that even though the `NOP` has a memory operand, when executed it does not access that memory in any way. This is simply how the `NOP` is represented in assembly code.

Anti-disassembly: multiple paths (2/2)

```
NOP WORD PTR [ESI-0x56FFFE45]       66 0F 1F 84 66 BB 01 00 A9
NOP WORD PTR [ECX+ESI-0x7F32BF40]   66 0F 1F 84 31 C0 40 CD 80
```

MEP vs HEP (Main Execution Path vs Hidden Execution Path)

```
MOV BX,0x0001                       66 BB 01 00
TEST EAX,0x841F0F66                 A9 66 0F 1F 84
XOR EAX,EAX                         31 C0
INC EAX                             40
INT 0x80                            CD 80
```

This corresponds to `exit(1)` in Linux x86 ABI.
Example taken from: [JLH13]

Overlapping instructions: not only for malware

Address	Byte	Sequence 1	Sequence 2	Sequence 3
454017	b8	mov eax, ebb907eb		
454018	eb			
454019	07			
45401a	b9			
45401b	eb		jmp 45402c	
45401c	0f	seto bl		
45401d	90			
45401e	eb			jmp 454028
45401f	08	or ch, bh		
454020	fd			

Figure 2: An example of overlapping instructions from a piece of malware. All three sequences of blocks execute.

Address	Byte	Sequence 1	Sequence 2
3fe9e8	74	je 3fe9eb	
3fe9e9	01		
3fe9ea	f0	lock cmpxchg %ecx, 0x35b0(%ebx)	
3fe9eb	0f		cmpxchg %ecx, 0x35b0(%ebx)
...	..		
3fe9f1	00		

Figure 3: An example of overlapping instructions from libc. The instruction starting at address 3fe9ea overlaps with the instruction starting at address 3fe9eb.

From: [MM16]

Demo/exercise

→`pony_580595`

- Check the entry point; can you fix the function-graph/decompiler views?
- The same "trick" is repeated 30 times, can you script your solution?
  - `pony_fix_anti_disasm.py` can get you started

Outline

Obfuscation

Obfuscation consists of transforming a program p into a functionally equivalent p', which is more difficult to analyze/understand

Obfuscated code can be larger and slower

- risk of introducing bugs
- difficult to test and debug

E.g., "Users have found that Denuvo, an anti-tamper program, causes slowdown, crashes, and freezes in legally purchased copies of the game"
https://www.gamerevolution.com/news/409231-sonic-mania-plus-drm-protection-slowing-down-legitimate-copies

Obfuscation classes

Data Hide/mask values by

- using different representations
- compressing/encrypting them
- replacing them with code
  (which calculates them at runtime)

Control-flow Hide the real control-flow

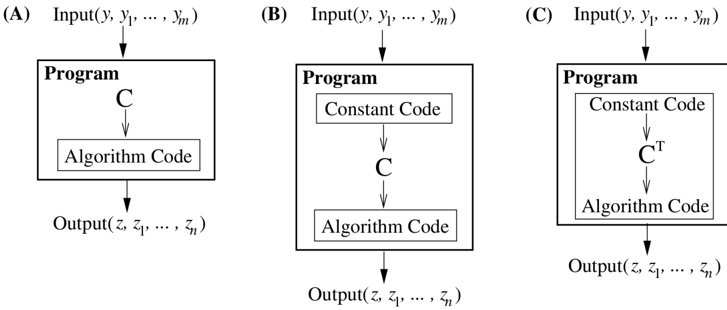
- behind irrelevant/superfluous code
- by introducing spurious ones
- by “removing” flows

Abstraction “Destroy” source-level abstractions by

- merging and splitting functions
- inlining library calls/directly invoking syscalls
- messing up calling-conventions

Outline

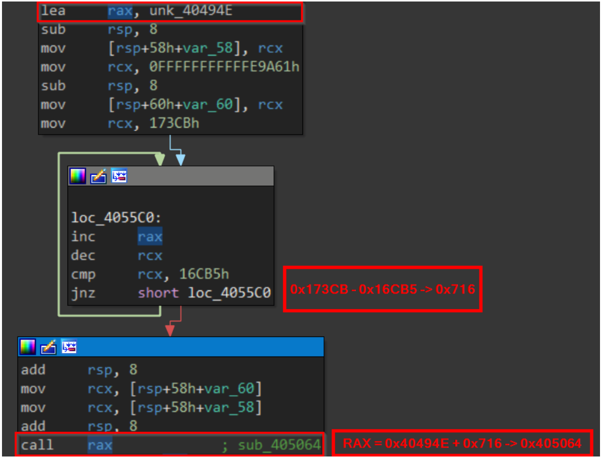
Data Obfuscation



- (a) both the constant C and the algorithm are in the clear
- (b) C is an *opaque constant*; i.e., C is replaced by code calculating C
- (c) C^T, i.e., C under transformation T, is created at runtime and the algorithm is modified to work with C^T (C is never directly exposed in memory); see, e.g., Mixed Boolean-Arithmetic obfuscation [ZMGJ07]

Opaque constant (and control-flow "removing") example

In a sample of the *Azov* ransomware, the direct call to `0x405064` has been replaced by an indirect one:



`https://research.checkpoint.com/2022/pulling-the-curtains-on-azov-ransomware-not-a-skidsware-but-polymorphic-wiper/`

Main techniques are:

- introducing irrelevant/superfluous code
- control-flow flattening
- opaque predicates
- virtualization

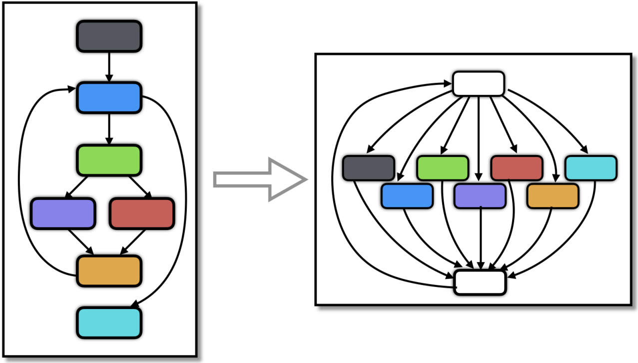
Example of introducing irrelevant code

Useless `sub`/`add` instructions in *Azov* ransomware:

```
and     rdx, 0
mov     edx, [rax]
mov     rax, [rbp+moduleBase]
sub     rax, 79C72h
add     rdx, rax
add     rdx, 79C72h
add     rax, 79C72h
mov     [rbp+addressOfNames], rdx
mov     rcx, [rbp+exportDirectory]
add     rcx, _IMAGE_EXPORT_DIRECTORY.AddressOfFunctions
xor     rdx, rdx
mov     edx, [rcx]
sub     rax, 1C5Eh
add     rdx, rax
add     rdx, 1C5Eh
add     rax, 1C5Eh
mov     [rbp+addressOfFunctions], rdx
```

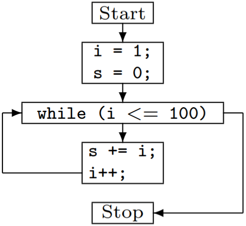
`https://research.checkpoint.com/2022/pulling-the-curtains-on-azov-ransomware-not-a-skidsware-but-polymorphic-wiper/`

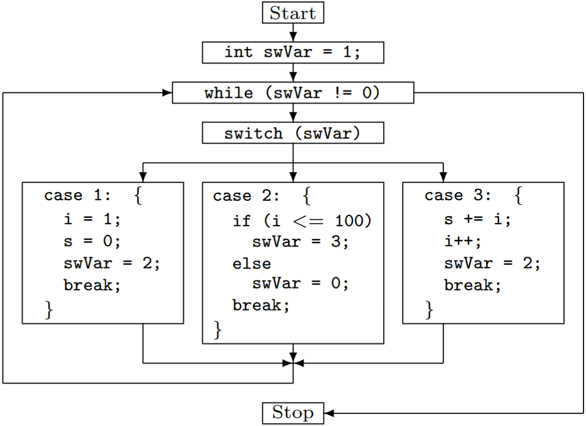
Control-flow flattening (1/2)



from https://tigress.wtf/flatten.html

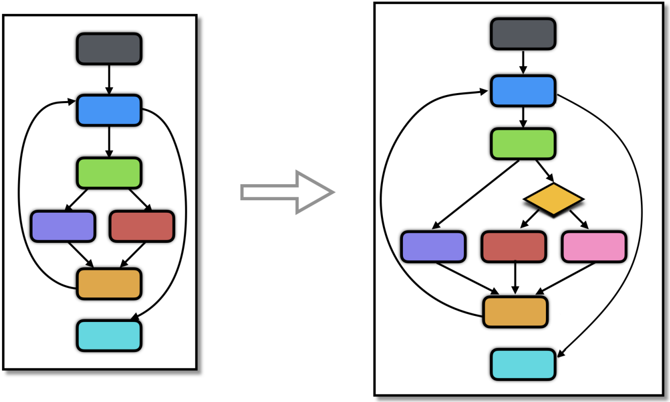
Control-flow flattening (2/2)





from [LK09]

Opaque predicates



from `https://tigress.wtf/addOpaque.html`

Opaque predicate examples (1/2)

Trivial:

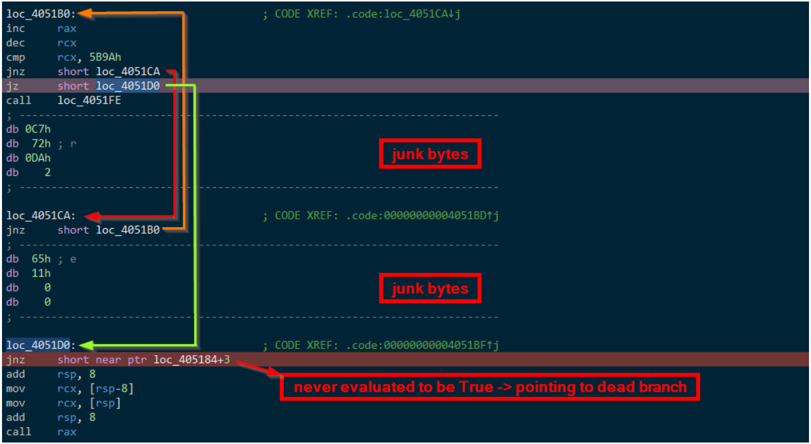
```
int x = 3;
int y = 1;
if ( (x - y) < 0) {
        // dummy code
} else {
        // real code
}
```

or

```
if (rand(1, 5) > 0){
        // real code
} else {
        // dummy code
}
```

Opaque predicate examples (2/2)

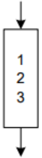
Opaque predicates and anti-disassembly (junk bytes) in *Azov* ransomware:

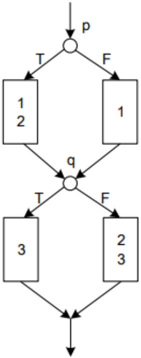


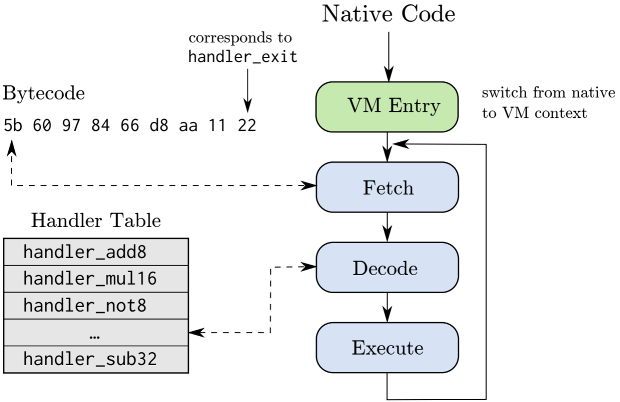
https://research.checkpoint.com/2022/pulling-the-curtains-on-azov-ransomware-not-a-skidsware-but-polymorphic-wiper/

Dynamic opaque predicates

Dynamic opaque predicates consists in using sets of correlated predicates, whose values is the same in one execution, but may change in another



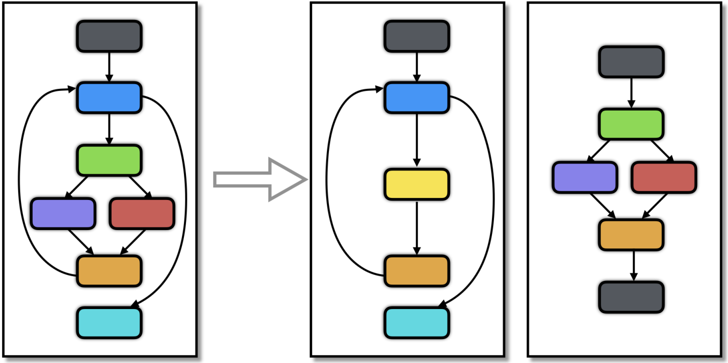




from [BCAH17]

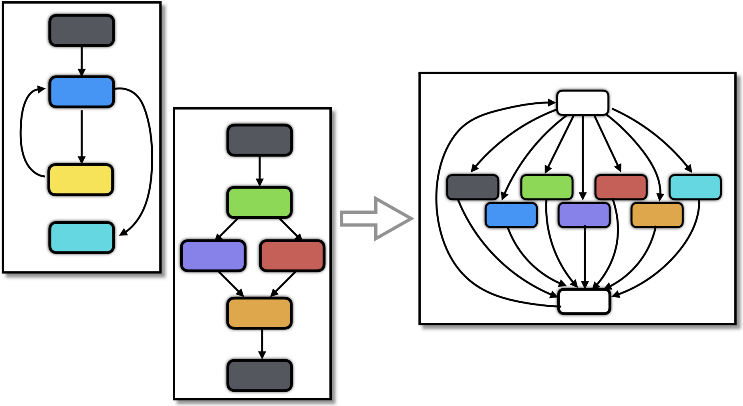
See `https://tigress.wtf/virtualize.html`

Function split



from https://tigress.wtf/split.html

Function merge (1/3)



from https://tigress.wtf/split.html

Function merge (2/3)

```
int foo(int x)
{
  return x*7;
}

void bar(int x, int z)
{
  if(x==z)
    printf("%d\n", x);
}

int main()
{
  int y = 6;
  y = foo(y);
  bar(y, 42);
}
```

could be transformed into...

Function merge (3/3)

```c
int foobar(int x, int z, int s)
{
  if(s==1)
    return x*7;
  else if(s==2)
    if(x==z)
      printf("%d\n", x);
  return 0;
}

int main()
{
  int y = 6;
  y = foobar(y, 99, 1);
  foobar(y, 42, 2);
}
```

Outline

Anti-Debugging techniques

As said, just *some* examples of techniques

- See https://anti-debug.checkpoint.com/ for a more complete, up-to-date, list
- Some code examples can be found on GitHub; e.g.
  - `https://github.com/bekdepo/AntiDBG`
  - `https://github.com/domin568/Anti-Debug-examples-Windows`

Debug flags

Flags in system tables indicate that a process is being debugged. These can be read by using API functions or examining memory

Windows API

- `kernel32!IsDebuggerPresent`
- `kernel32!CheckRemoteDebuggerPresent`
- ...

Manual checks, by reading:

- `BeingDebugged`, byte at offset 2 in the PEB (PEB = `fs:[0x30]`)
- `NtGlobalFlag` in the PEB
- Heap flags
- ...

`https://www.nirsoft.net/kernel_struct/vista/TEB.html`

`https://www.nirsoft.net/kernel_struct/vista/PEB.html`

Linux

- API: `ptrace`
- Manual checks: `TracerPid` in `/proc/self/status`

Parent processes

Another simple check consists in verifying the parent-name/presence of "suspicious" windows; e.g. the names of known debugging tools

- Windows: `user32!FindWindow(Ex)`, `user32!EnumWindows`, `user32!EnumThreadWindows`, ...
- Linux: You can get the parent via `getppid`, and then check `Name` in `/proc/`*`parent-pid`*`/status`

To thwart string analysis, known-names can be hashed/encrypted

Hiding from the debugger

`ntdll!NtSetInformationThread` can hide a thread from a debugger

- by passing the undocumented value `THREAD_INFORMATION_CLASS::ThreadHideFromDebugger` (0x11)
- After the thread is hidden, the debugger won't receive events

```
#define NtCurrentThread ((HANDLE)-2)

bool AntiDebug()
{
    NTSTATUS status = ntdll::NtSetInformationThread(
        NtCurrentThread,
        ntdll::THREAD_INFORMATION_CLASS::ThreadHideFromDebugger,
        NULL,
        0);
    return status >= 0;
}
```

https://anti-debug.checkpoint.com/techniques/interactive.html

Exceptions are handled differently

- `kernel32!CloseHandle`/`ntdll!NtClose` raise an exception when an invalid handle is passed *and* the process is being debugged
  https://docs.microsoft.com/en-us/windows/win32/api/handleapi/nf-handleapi-closehandle
- `kernel32!SetUnhandledExceptionFilter` allows an application to replace the default handler; however, the handler is not called is the process is being debugged
  https://docs.microsoft.com/en-us/windows/win32/api/errhandlingapi/nf-errhandlingapi-setunhandledexceptionfilter
- `kernel32!RaiseException` can be used to raise exceptions consumed by a debugger, such as `DBC_CONTROL_C`
  https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-exception_record

Unix Signals

Similar idea, with signals:

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void handler(int signo)
{
        printf("Not debugged!\n");
        exit(0);
}

int main()
{
        signal(SIGTRAP, handler);
        asm("int3");
        printf("Debugged...\n");
}
```

Timing

General idea: measuring the delay introduced by user-interactions

1. get time
2. do some work
3. calculate difference

e.g. by using `RDTSC`, `kernel32!GetTickCount`, ...

Breakpoints

SW BP can

- be searched/removed in
  - pre-determined memory ranges (e.g., function bodies) or
  - runtime locations; e.g., a function can retrieve its saved return address and look for a BP at that instruction
- make the program crash if routines are copied to newly allocated memory and run there

HW BP can be discovered through `GetThreadContext`

Self-debugging

Linux As already mentioned, `ptrace(PTRACE_TRACEME, ...)`:

- if it fails, there is a debugger
- OTOH if it succeeds twice (or when called with bad arguments), then there are probably both a debugger *and* `ptrace` has been hooked
  - A program may check `LD_PRELOAD` too

Windows You can create a child process that tries to debug its parent

Mitigations

- Skipping/patching the checks
- Flags in user-space (e.g. `BeingDebugged`) can be altered
- Functions that invoke system-calls can be *intercepted*
  - breakpoints
  - hooking
  - ...

Anti anti-debug: ScyllaHide

ScyllaHide, `https://github.com/x64dbg/ScyllaHide`, can automatically defeat most anti-debugging techniques

Demo/exercises

→`bmatter_22d7d6` (it must be renamed into `....exe`)
→`s0_xCCited`, a Linux CTF-challenge by Lorenzo Maffia

Outline

Environment detection

Sandboxes and other analysis environment can be detected by checking

- the presence of a debugger/special processes/loaded DLLs
- the number of running processes
- installed programs/their absence
- the presence of an hypervisor
- RAM size/screen resolution/...
- mouse movement, keyboard activity, ...
- ...

Various open-source projects contain example code for detection:

al-khaser `https://github.com/LordNoteworthy/al-khaser`

Pafish `https://github.com/a0rtega/pafish`

Let's discuss some examples...

Specific hardware/drivers

- Virtual network interfaces can have specific MACs
  - e.g., VMware has `00-50-56-...`, `00-0C-29-...`, ... See, e.g., `https://hwaddress.com/company/vmware-inc/`
- Display device names may contain the string `"VMware"`; `https://twitter.com/Maff1t/status/1618556982952091649`
- Drivers, found in the
  - registry: `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Class \{4D36E968-E325-11CE-BFC1-08002BE10318}\0000\DriverDesc`
  - file system: `%WINDIR%\system32\drivers\vmmouse.sys`
- by using `CPUID` — `https://kb.vmware.com/s/article/1009458`

```
BOOL is_vm() {
        INT CPUInfo[4] = { -1 };
        __cpuid(CPUInfo, 1);
        return ((CPUInfo[2] >> 31) & 1);
}
```

- ...

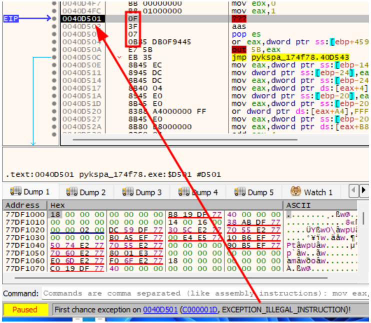
Special instructions

- Virtual PC interprets `0F 3F 07 0B` as `vpcext 7, 0Bh` (illegal instruction in other environments)
- VMware uses the I/O port `0x5658` ("VX" in ASCII)

```
mov EAX, 564D5868h ; "VMXh"
xor EBX, EBX       ; set EBX to anything but VMXh
mov CX, 0Ah        ; command 10: Get VMware version
mov DX, 5658h      ; VX
in EAX, DX
cmp EBX, 564D5868h ; EBX should have the magic number
                   ; if VMware is present
```

~~Demo~~ (it doesn't seem to work on latest Windows)

→`pykspa_174f78` (it must be renamed into `....exe`)



It does not seem to work properly on Windows 11+
(otherwise, patching `0x40dbee` with `sub eax, eax` should be enough)

Outline

Introduction

Packers are programs that transform executables

- originally, used to save (disk) space by compressing programs
- nowadays, mostly used to thwart static analysis
  - they may compress, encrypt, add anti-debug/VM checks, ...

When an executable gets packed:

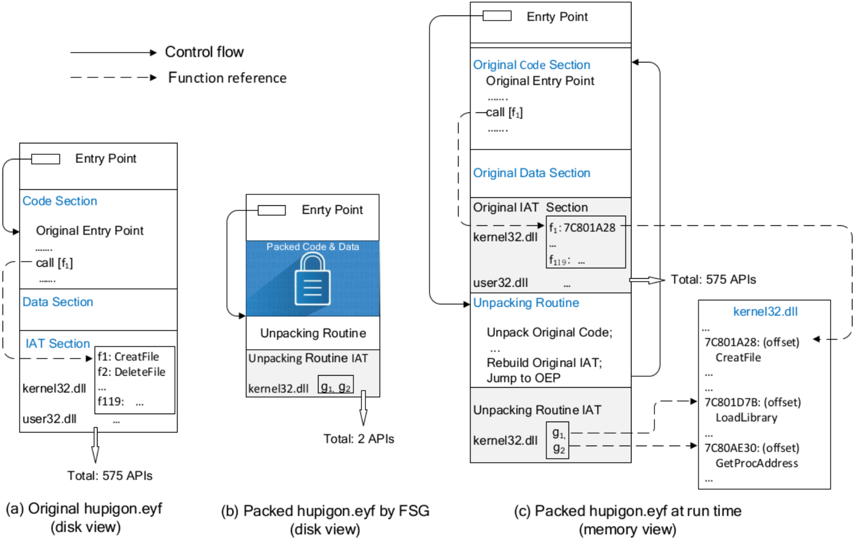
- an unpacking stub is added to the packed program, whose entry-point points to this stub
- (most) references to external libraries/functions are typically removed

Unpacking stub

The unpacking stub is a piece of code that

1. unpacks (=decompress/decrypt) the original program
   - in the same process, or in others to evade detection
   - often, it needs to allocate new memory
2. usually, resolves (the original) imports
   - loading the corresponding libraries if needed
3. jumps to the Original Entry Point (OEP), with the so-called tail-jump

Example: FSG



Taken from [CMF$^+$18]

Indicators of packing

How can we detect a packed executable?

- few imports, in particular `LoadLibrary` and `GetProcAddress`
- abnormal section features, like
  - names, e.g., `UPX0`/`UPX1`
  - sizes; e.g., an empty code section (on disk, with non-zero virtual size)
- entropy is an indicator, but it may be unreliable [MAUP+20]

Programs like

- Detect It Easy — `https://github.com/horsicq/Detect-It-Easy`
- *PE Bear* — `https://github.com/hasherezade/pe-bear`
- *PE Studio* — `https://www.winitor.com/`

can help

Unpacking options

Commodity packers, e.g. UPX, may provide an unpacker.

If an unpacker is not available, we can tackle the problem by

- reversing the unpacking algorithm, and then writing an unpacker
- running/emulating the program until it is unpacked (in some "jail/sandbox", using VMs, emulators, instrumentation, ...) to
  1. Find the OEP
  2. Dump the unpacked version from memory

  Then, fix/reconstruct the PE header. See:
  - Scylla, integrated in x64dbg
    `https://github.com/NtQuery/Scylla`
  - *Mal-Unpack*
    `https://github.com/hasherezade/mal_unpack`
  - *Unipacker*
    `https://github.com/unipacker/unipacker`

Finding the OEP and dumping the program

Typically, the unpacking stub

1. Saves the execution-context on the stack
2. Unpacks its payload
3. Resolve the imports
   - An API call from a rebuilt-IAT means that the control flow already reached the OEP [CMF+18]
   - A BP/hook on `GetProcAddress` may allow you to detect the beginning of this phase
4. Restores the original context, by "popping" it from the stack
   - A HW-breakpoint on stack access can help in finding the OEP
5. Jumps to the OEP
   - At the beginning, the area containing the OEP is usually invalid/empty
   - Execution of addresses written by the program can indicate the OEP (or another layer of packing [UPBSB15])

Because of standard initialization code, programs may

- share the same startup code, which can be searched in memory
- call common functions, e.g. `GetCommandLine`; BPs on those can help

Special cases

If the unpacker is "lazy" and relies on standard API `RtlDecompressBuffer` and/or `CryptDecrypt`:

- BP on those functions
- When BP is hit, get the
  - address of the buffer
  - pointer to the (output) size

  then execute until return
- Check/dump the memory area

A general approach using a debugger

Breakpoint on

RtlDecompressBuffer	
CryptDecrypt	(see previous slide)
CreateProcess{*,InternalW}	used in many *injection* techniques; however, `ShellExecute` uses flag `CREATE_SUSPENDED`
NtResumeThread	and
NtUnmapViewOfSection	used by *process hollowing*
VirtualProtect(Ex)	*hooking* or dynamically-generated code
LocalAlloc/VirtualAlloc(Ex)	the result value can be followed in dump to see if something interesting is written there
NtWriteVirtualMemory	
WriteProcessMemory	to see what gets written, and where
CreateRemoteThread	to start shellcode/`LoadLibrary`
CreateFileTransactedW	may indicate process Doppleganging

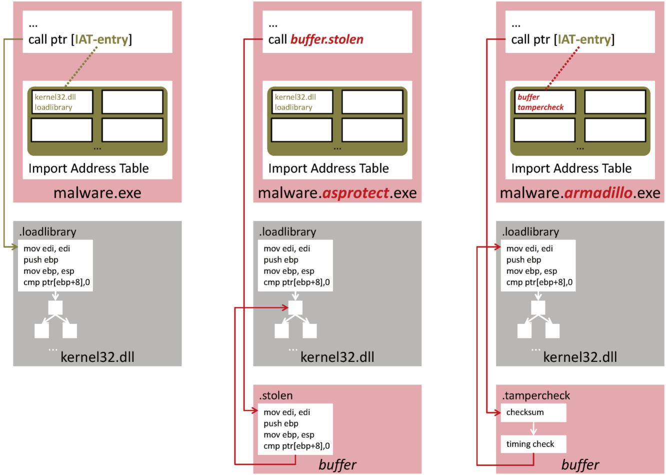
See also `https://www.elastic.co/blog/ten-process-injection-techniques-technical-survey-common-and-trending-process`

Beware: alternative ways to read/write process memory

- *GetEnvironmentVariable as an alternative to WriteProcessMemory in process injections*
  `https://x-c3ll.github.io/posts/GetEnvironmentVariable-Process-Injection/`
- *Reading and writing remote process data without using ReadProcessMemory/WriteProcessMemory*
  `https://www.x86matthew.com/view_post?id=read_write_proc_memory`
- ...

Anti-hooking techniques

To circumvent patch-based tracing, some packers employ the *stolen bytes* technique, others load a new copy of `ntdll`/overwrite the hooked one



Taken from [RM13]

Demo/exercises

(Too ☺) Classic injection	→`inj32to64_33e0e9`
Anti-debug/Unpacking	→`notepad*exe`
?	→`basic_calc*exe`

References I

[BCAH17] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz.
Syntia: Synthesizing the semantics of obfuscated code.
In *26th USENIX Security Symposium (USENIX Security 17)*, pages 643–659, 2017.

[CMF+18] Binlin Cheng, Jiang Ming, Jianmin Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion.
Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with orders-of-magnitude performance boost.
In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 395–411, 2018.

References II

[JLH13] Christopher Jamthagen, Patrik Lantz, and Martin Hell.
A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries.
In *Anti-malware Testing Research (WATeR), 2013 Workshop on*, pages 1–9. IEEE, 2013.

[LK09] Tımea László and Ákos Kiss.
Obfuscating c++ programs via control flow flattening.
Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica, 30(1):3–19, 2009.

[MAUP+20] Alessandro Mantovani, Simone Aonzo, Xabier Ugarte-Pedrero, Alessio Merlo, and Davide Balzarotti.
Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem.
In *NDSS*, 2020.

References III

[MM16] Xiaozhu Meng and Barton P Miller.
Binary code is not easy.
In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 24–35. ACM, 2016.

[RM13] Kevin A Roundy and Barton P Miller.
Binary-code obfuscations in prevalent packer tools.
ACM Computing Surveys (CSUR), 46(1):1–32, 2013.

[UPBSB15] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas.
SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers.
In *2015 IEEE Symposium on Security and Privacy*, pages 659–673. IEEE, 2015.

References IV

[ZMGJ07] Yongxin Zhou, Alec Main, Yuan X. Gu, and Harold Johnson.
Information Hiding in Software with Mixed Boolean-Arithmetic Transforms.
In *Proceedings of the 8th International Conference on Information Security Applications*, WISA'07, page 61–75, Berlin, Heidelberg, 2007. Springer-Verlag.