



IT Security

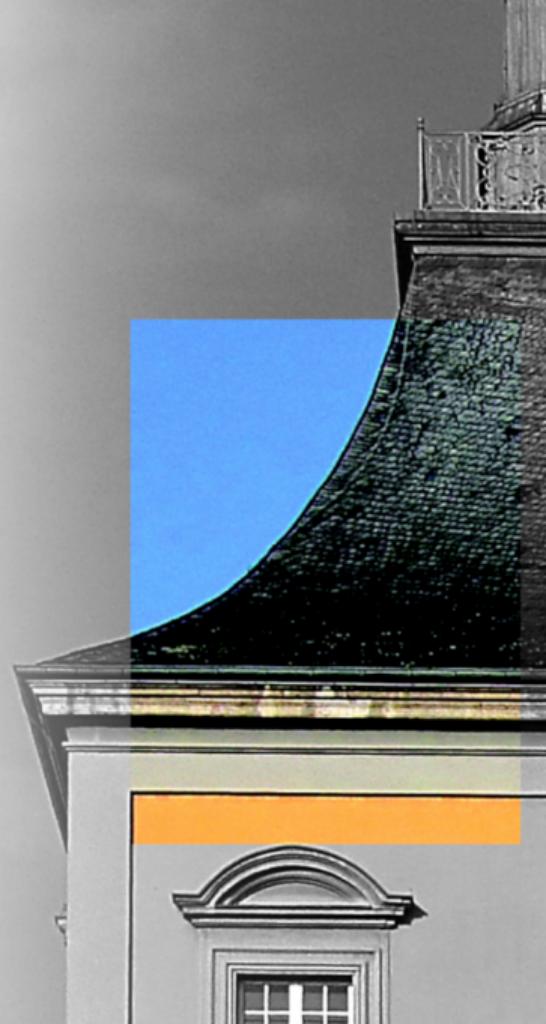
Malware Analysis

Timo Pohl

pohl@cs.uni-bonn.de

University of Bonn | Institute of Computer Science 4

Lecture IT Security | Uni Bonn | WT 2024/25



Contents

-  Motivation
-  Getting Infected
 - PDF Analysis
-  PE Analysis
 - Obfuscation
 - Anti-Debugging
 - Anti-Sandboxing
-  Outlook

Motivation

- There is a full module on malware analysis
- This is a speedrun through some interesting topics
- If you're interested, visit the module

Why Malware Analysis?

Cyber Attack: Targobank Blocks access to thousands of accounts

After attempted access by malicious players to online banking, Targobank blocked the access of thousands of customers. New access data is created

Reading time: 2 min.  Save in Pocket

DP World: Safety incident paralyses key port operators in Australia

A cybersecurity incident has paralysed the port operator DP World Australia. The government considers the situation to be "very serious".

Reading time: 2 min.  Save in Pocket

   8



After ransomware attack: South Westphalia IT and municipalities refuse ransom payment

The municipal IT service provider Südwestfalen-IT rejects the payment of a ransom money after a ransomware attack in consultation with the affected municipalities.

Reading time: 2 min.  Save in Pocket

13.11.2023 1
By Dirk Knop



(Image: Pixels Hunter/Shutterstock.com)

Dena: Cyber attack on the German Energy Agency

The Dena has been incapacitated since Sunday afternoon – due to a cyber attack, as the Bund-owned GmbH announced on Tuesday afternoon.

Reading time: 2 min.  Save in Pocket

   66.66



(Image: Dena)

Why Malware Analysis?

- Find out how malware works
- Stop ongoing campaigns
- Develop countermeasures

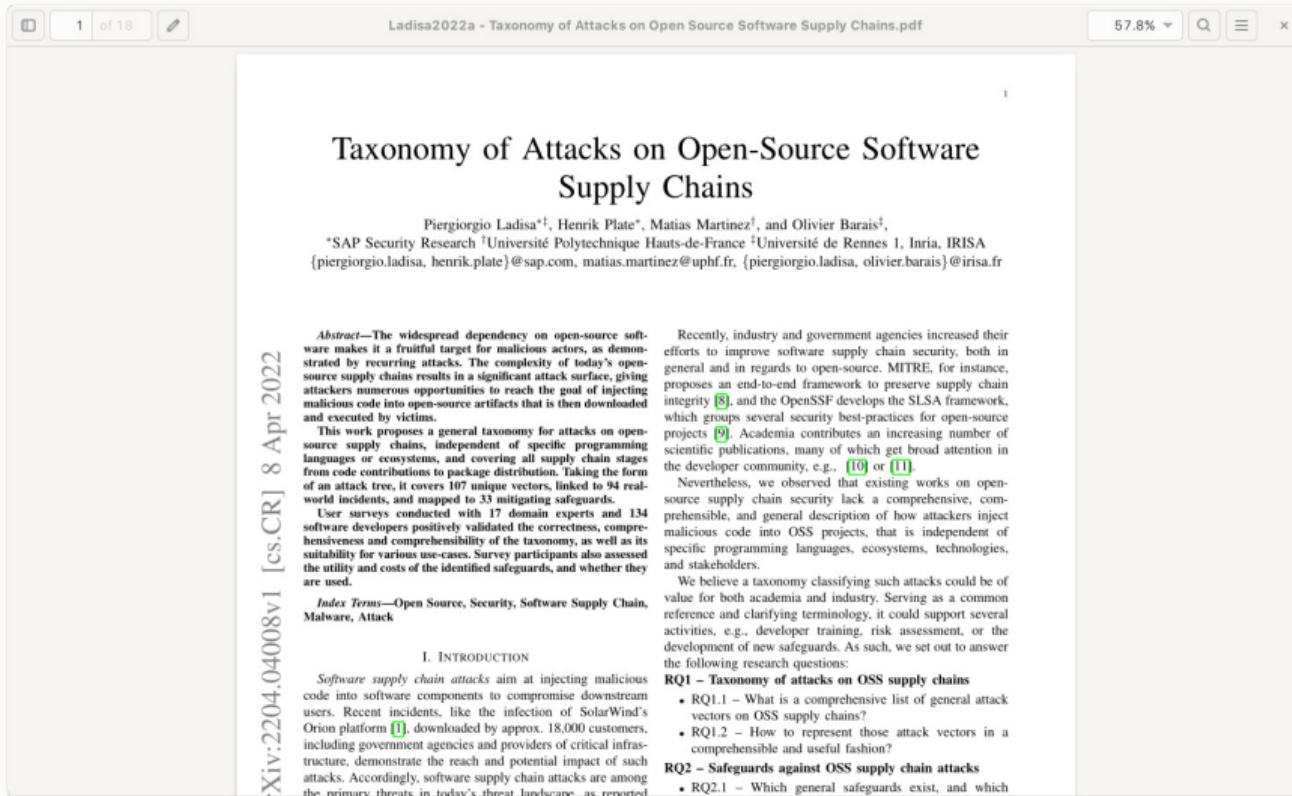
Getting Infected

-  Motivation
-  Getting Infected
 - PDF Analysis
-  PE Analysis
 - Obfuscation
 - Anti-Debugging
 - Anti-Sandboxing
-  Outlook

Getting infected

- PDFs
- Phishing
- Malicious updates
- Office macros
- ...

PDFs show Documents!



Ladisa2022a - Taxonomy of Attacks on Open Source Software Supply Chains.pdf 57.8% x

Taxonomy of Attacks on Open-Source Software Supply Chains

Piergiorgio Ladisa<sup>*, Henrik Plate^{*}, Matias Martinez[†], and Olivier Barais[‡],
*SAP Security Research [†]Université Polytechnique Hauts-de-France [‡]Université de Rennes 1, Inria, IRISA
{piergiorgio.ladisa, henrik.plate}@sap.com, matias.martinez@uphf.fr, {piergiorgio.ladisa, olivier.barais}@irisa.fr</sup>

Abstract—The widespread dependency on open-source software makes it a fruitful target for malicious actors, as demonstrated by recurring attacks. The complexity of today's open-source supply chains results in a significant attack surface, giving attackers numerous opportunities to reach the goal of injecting malicious code into open-source artifacts that is then downloaded and executed by victims.

This work proposes a general taxonomy for attacks on open-source supply chains, independent of specific programming languages or ecosystems, and covering all supply chain stages from code contributions to package distribution. Taking the form of an attack tree, it covers 107 unique vectors, linked to 94 real-world incidents, and mapped to 33 mitigating safeguards.

User surveys conducted with 17 domain experts and 134 software developers positively validated the completeness, comprehensiveness and comprehensibility of the taxonomy, as well as its suitability for various use-cases. Survey participants also assessed the utility and costs of the identified safeguards, and whether they are used.

Index Terms—Open Source, Security, Software Supply Chain, Malware, Attack

I. INTRODUCTION

Software supply chain attacks aim at injecting malicious code into software components to compromise downstream users. Recent incidents, like the infection of SolarWind's Orion platform [1], downloaded by approx. 18,000 customers, including government agencies and providers of critical infrastructure, demonstrate the reach and potential impact of such attacks. Accordingly, software supply chain attacks are among the primary threats in today's threat landscape, as reported

Recently, industry and government agencies increased their efforts to improve software supply chain security, both in general and in regards to open-source. MITRE, for instance, proposes an end-to-end framework to preserve supply chain integrity [8], and the OpenSSF develops the SLSA framework, which groups several security best-practices for open-source projects [9]. Academia contributes an increasing number of scientific publications, many of which get broad attention in the developer community, e.g., [10] or [11].

Nevertheless, we observed that existing works on open-source supply chain security lack a comprehensive, comprehensible, and general description of how attackers inject malicious code into OSS projects, that is independent of specific programming languages, ecosystems, technologies, and stakeholders.

We believe a taxonomy classifying such attacks could be of value for both academia and industry. Serving as a common reference and clarifying terminology, it could support several activities, e.g., developer training, risk assessment, or the development of new safeguards. As such, we set out to answer the following research questions:

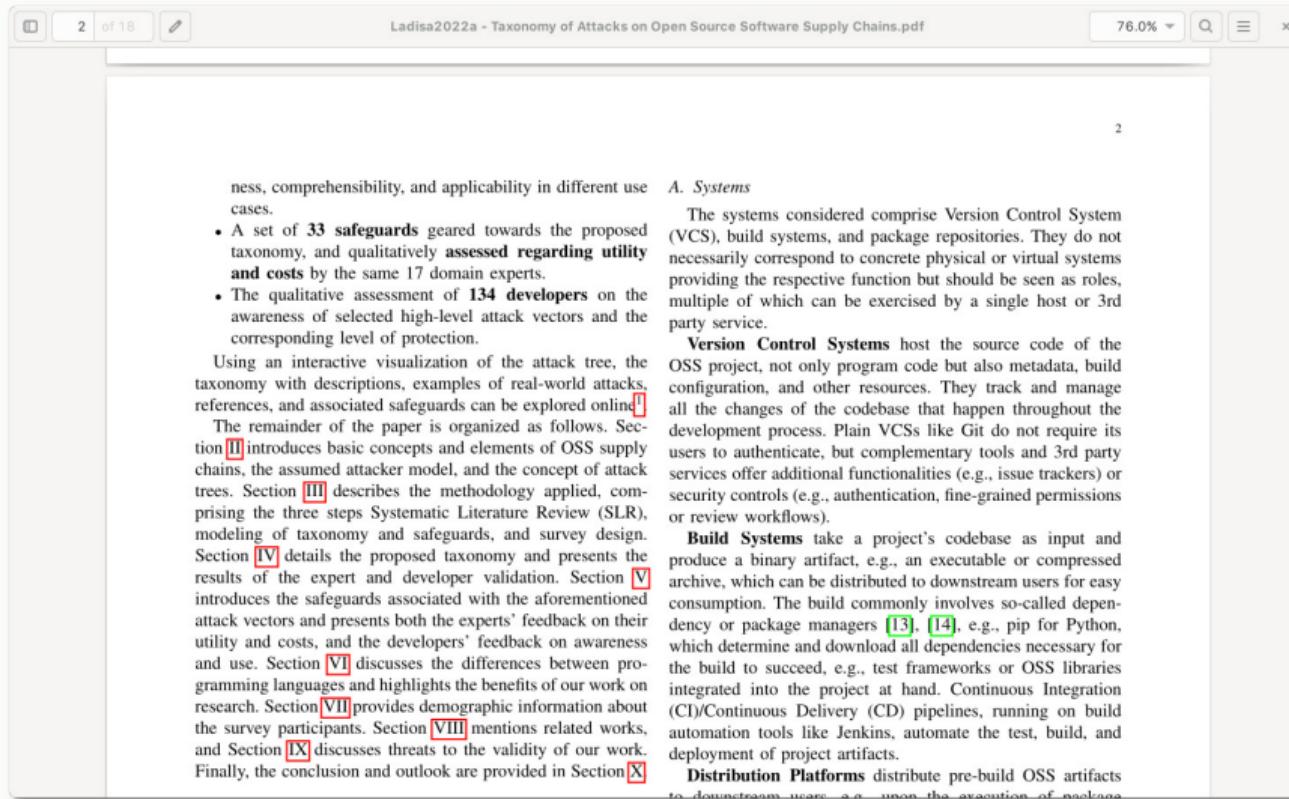
RQ1 – Taxonomy of attacks on OSS supply chains

- RQ1.1 – What is a comprehensive list of general attack vectors on OSS supply chains?
- RQ1.2 – How to represent those attack vectors in a comprehensible and useful fashion?

RQ2 – Safeguards against OSS supply chain attacks

- RQ2.1 – Which general safeguards exist, and which

PDFs are interactive.



Ladisa2022a - Taxonomy of Attacks on Open Source Software Supply Chains.pdf
76.0% 2

ness, comprehensibility, and applicability in different use cases.

- A set of **33 safeguards** geared towards the proposed taxonomy, and qualitatively **assessed regarding utility and costs** by the same 17 domain experts.
- The qualitative assessment of **134 developers** on the awareness of selected high-level attack vectors and the corresponding level of protection.

Using an interactive visualization of the attack tree, the taxonomy with descriptions, examples of real-world attacks, references, and associated safeguards can be explored online [1].

The remainder of the paper is organized as follows. Section [II] introduces basic concepts and elements of OSS supply chains, the assumed attacker model, and the concept of attack trees. Section [III] describes the methodology applied, comprising the three steps Systematic Literature Review (SLR), modeling of taxonomy and safeguards, and survey design. Section [IV] details the proposed taxonomy and presents the results of the expert and developer validation. Section [V] introduces the safeguards associated with the aforementioned attack vectors and presents both the experts' feedback on their utility and costs, and the developers' feedback on awareness and use. Section [VI] discusses the differences between programming languages and highlights the benefits of our work on research. Section [VII] provides demographic information about the survey participants. Section [VIII] mentions related works, and Section [IX] discusses threats to the validity of our work. Finally, the conclusion and outlook are provided in Section [X].

A. Systems

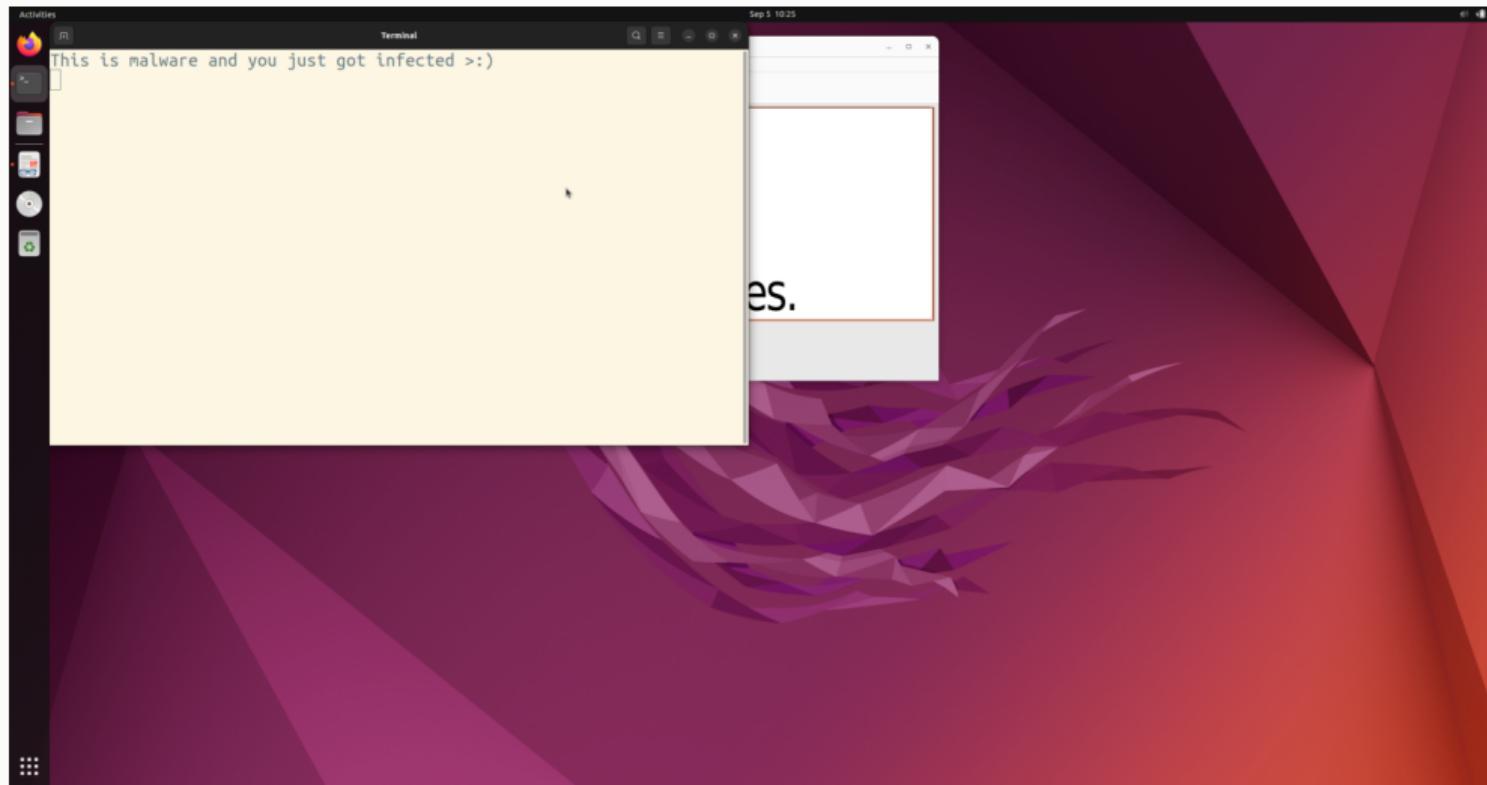
The systems considered comprise Version Control System (VCS), build systems, and package repositories. They do not necessarily correspond to concrete physical or virtual systems providing the respective function but should be seen as roles, multiple of which can be exercised by a single host or 3rd party service.

Version Control Systems host the source code of the OSS project, not only program code but also metadata, build configuration, and other resources. They track and manage all the changes of the codebase that happen throughout the development process. Plain VCSs like Git do not require its users to authenticate, but complementary tools and 3rd party services offer additional functionalities (e.g., issue trackers) or security controls (e.g., authentication, fine-grained permissions or review workflows).

Build Systems take a project's codebase as input and produce a binary artifact, e.g., an executable or compressed archive, which can be distributed to downstream users for easy consumption. The build commonly involves so-called dependency or package managers [13], [14], e.g., pip for Python, which determine and download all dependencies necessary for the build to succeed, e.g., test frameworks or OSS libraries integrated into the project at hand. Continuous Integration (CI)/Continuous Delivery (CD) pipelines, running on build automation tools like Jenkins, automate the test, build, and deployment of project artifacts.

Distribution Platforms distribute pre-build OSS artifacts to downstream users, e.g., upon the execution of package

PDFs... run code?



PDFs are (mostly) readable

```
1 %PDF-1.5
2 1 0 obj
3 <</Type/Pages/Kids[8 0 R]/Count 1/Resources 3 0 R/MediaBox[0 0 700 200]>>
4 endobj
5 2 0 obj
6 <</Type/Font/Subtype/Type1/BaseFont/Noto#20Sans>>
7 endobj
8 3 0 obj
9 <</Font<</F1 2 0 R>>>>
10 endobj
11 4 0 obj
12 <</Length 60>>stream
```

- Large collection of objects
- Mostly readable
- Direct objects like 1, (hello) or [null true false]
- Indirect objects starting with <ID> <Gen-No> obj and ending with endobj

PDF Structure Example

```
1 %PDF-1.5
2 1 0 obj
3 <</Type/Pages/Kids[8 0 R]/Count 1/Resources 3 0 R/MediaBox[0 0 700 200]>>
4 endobj
5 2 0 obj
6 <</Type/Font/Subtype/Type1/BaseFont/Noto#20Sans>>
7 endobj
8 3 0 obj
9 <</Font<</F1 2 0 R>>>>
10 endobj
11 4 0 obj
12 <</Length 60>>stream
```

Common Object Types

- Name objects — /Name, /anything
- Reference objects — 8 0 R
- Strings — (as literals) or <68656c6c6f> hex encoded
- Numbers — 0 or 123994
- Arrays — [(hello) /world]
- Dictionaries — <</Key1 /AnyObj /Key2 8 0 R>>

PDF Types Example

```
3 <<
4   /Type /Pages
5   /Kids [8 0 R]
6   /Count 1
7   /Resources 3 0 R
8   /MediaBox [0 0 700 200]
9   /MadeUpKey ("String literal in here!")
10 >>
```

Start of a PDF

```
1 trailer
2 <<
3   /Size 713
4   /Root 711 0 R
5   /Info 712 0 R
6 >>
7 startxref
8 281353 <- Byte offset of "start" dict
9 %%EOF
```

```
1 10 0 obj
2 <<
3   /Root 9 0 R
4   /Type /XRef
5   /Size 11
6   /W [1 4 2]
7   /Index [1 10]
8   /Length 70
9 >>
10 endobj
11
12 startxref
13 816
14 %%EOF
```

PDF Interactive Features

PDF Action Dictionaries

```
23 6 0 obj
24 <<
25     /Type /Action
26     /S /Launch
27     /F (/tmp/malware.py)
28 >>
29 endobj
30 7 0 obj
31 <<
32     /Type /Action
33     /S /JavaScript
34     /JS (this.exportAsText(true, '/tmp/file.txt');)
35 >>
```

PDF Interesting Actions

- [/Launch](#)
 - [/JavaScript](#)
 - [/SubmitForm](#)
 - [/URI](#)
- Launch an application. Execute JavaScript. Send data to a specified URL. Open a URI.

PDF OpenAction

```
1 9 0 obj
2 <<
3   /Type /Catalog
4   /Pages 1 0 R
5   /OpenAction 5 0 R
6 >>
```

PDF OpenAction

- PDFs have Catalog dictionaries
- Catalog dictionary has “content catalog”
- OpenAction defined in Catalog dictionary
- OpenAction executed when opening document

- We start at byte offset 754
- Points at object 10 0
- Root dict points at object 9 0

PDF Full Example

```
50 9 0 obj
51   <<
52     /Type /Catalog
53     /Pages 1 0 R
54   >>
55 endobj
```

PDF Full Example

- Object 9 0 is Catalog dictionary
- Has pages object at 1 0

PDF Full Example

```
1 %PDF-1.5
2 1 0 obj
3   <<
4     /Type /Pages
5     /Kids [8 0 R]
6     /Count 1
7     /Resources 3 0 R
8     /MediaBox [0 0 700 200]
9   >>
10 endobj
```

PDF Full Example

- Pages object shows one page object at 8 0
- Resources object 3 0 is irrelevant for us

PDF Full Example

```
33 8 0 obj
34     <<
35         /Type /Page
36         /Parents 1 0 R
37         /Contents 4 0 R
38         /Annots [
39             <<
40                 /Type /Annot
41                 /Subtype /Link
42                 /Open true
43                 /A 6 0 R
44                 /H /N
45                 /Rect [0 0 700 200]
46             >>
47         ]
48     >>
49 endobj
```

PDF Full Example

- Parents object is where we come from
- Contents object at 4 0 defines page content
- Annots object defines annotations
- Annotation defines an action at 6 0

PDF Full Example

```
12 4 0 obj
13   <<
14     /Length 60
15   >>
16   stream
17     BT
18     /F1 48 Tf
19     0 10 Td
20     (Click me for free cat pictures.) Tj
21     ET
22   endstream
23 endobj
```

PDF Full Example

- Content object has no relevant information for us
- We see the displayed text in the byte stream

PDF Full Example

```
25 6 0 obj  
26 <<  
27     /Type /Action  
28     /S /Launch  
29     /F (/tmp/malware.py)  
30 >>
```

- Annotation action is indeed an action dict
- Contains a Launch action
- Launches /tmp/malware.py

PDFs can be “obfuscated”

- Different string representation
- Stream compression
- Object encryption

```
1 6 0 obj
2 <<
3   /Type /Action
4   /S /Launch
5   /F (/tmp/malware.py)
6 >>
7 endobj
8 6 0 obj
9 <<
10  /Type /Action
11  /S /Launch
12  /F <2F746D702F6D616C776172652E7079>
13 >>
14 endobj
```

PDF analysis

- Your favourite text editor
- qpdf for “deobfuscation”
- peepdf for a quick overview

```
(kali㉿kali)-[~/peepdf]
$ python2 peepdf.py ~/one-hundred-percent-benign.pdf
Warning: PyV8 is not installed!!
Warning: pylibemu is not installed!!
Warning: Python Imaging Library (PIL) is not installed!!

File: one-hundred-percent-benign.pdf
MD5: a1fb993f95c3fdf012061f7b8a4fc64a
SHA1: 0fe21d975fd03f571b41e9ae1546e9fce24427ea
SHA256: 9e023fb09207ac61a1bb0f389467a9f9d2d1242be461385814628ba1ce8c88e5
Size: 1025 bytes
Version: 1.5
Binary: False
Linearized: False
Encrypted: False
Updates: 0
Objects: 10
Streams: 2
URIs: 1
Comments: 0
Errors: 1

Version 0:
    Catalog: 9
    Info: No
    Objects (10): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    Streams (2): [4, 10]
        Xref streams (1): [10]
        Encoded (0): []
    Objects with URIs (1): [5]
    Objects with JS code (1): [7]
    Suspicious elements:
        /OpenAction (1): [9]
        /JavaScript (1): [7]
        /Launch (1): [6]
```

Summary

- PDFs are readable as text
- PDFs can perform actions
- Analysis is relatively simple
- Most mainstream PDF viewers are secure against common attacks

Questions?

-  Motivation
-  Getting Infected
 - PDF Analysis
-  PE Analysis
 - Obfuscation
 - Anti-Debugging
 - Anti-Sandboxing
-  Outlook

Comparison to ELF Analysis

- ELF analysis known from previous lecture
- Windows malware is most prevalent
- Windows uses PE instead of ELF
- stdcall and Microsoft x64 instead of cdecl or System V
- x64dbg instead of gdb
- Similar tools for static analysis (ghidra, ida, binja, ...) + some for quick overviews (PE Studie, Detect it Easy, ...)
- Focus on strings, network access, ... — we want to understand behaviour

General approach

- Goal: understand malware behaviour
- Top-Down approach: start at entry point, follow control flow
- Bottom-Up approach: search for interesting things like API calls, strings, encryption routines, ... and backtrace from there

Obfuscation

-  Motivation
-  Getting Infected
 - PDF Analysis
-  PE Analysis
 - Obfuscation
 - Anti-Debugging
 - Anti-Sandboxing
-  Outlook

What is Obfuscation?

“carlini” from the 27th IOCCC

```

1 int*d,D[9999],N=20,L=4,n,m,k,a[3],i;char*p,*q,S[2000]={"L@X-SGD-HNBBB-AD-VHSG-\
2 -XNT\x1b[2J\x1b[H",*s=S,*G="r2zZX!.+@KBK^yh.!:%Bud!.+Jyh.!6.BHBhp!6.BHBh!:%Bu\
3 v{VT!.hBJ6p!042ljn!284b}`!.hR6Dp!.hp!h.T6p!h.p6!2/LilqP72!h.+@QB!~}lqP72/Lil!\
4 h.+@QBfp!:)0?F]nwf!,82v!.sv{6!.l6!,j<n8!.xN6t!&NvN*!.6hp";/*Stay_on_target.*/
5 #include/**/<complex.h>/****/*Oh,my_dear_friend.How_I've_missed_you.--C-3PO***/\n
6 typedef/**/complex/**/double(c);c(X)[3],P,O;c/**/B(double t){double s=1-t,u=P=\n
7 s*s*X[1]           +2          *s*t*          *X+t          *t*X          [2]+0;u=I*P;\n
8 return+48*((      s=P)+   48*I )/(    1<u?    u:    1);} /* 1977 IOCCC2020*/\n
9 #include/**/ Do.Or do_not . There_is_ no_try... --Yoda**/<stdio.h>\n
10 void/**/b( double t,/**/ * **/double u){double s=P=B(t)-B(u);(s=P\n
11 *(2*s-P)) <1?m=P=B ((t+ u)/ 2),k =- I*P, m> -4188m<39889<k88k\n
12 <48?           m+=k/        2*80+       73,S [m]=           S[m]\n
13 -73?k%2?S[m]-94?95:73:S[m]-95?94:73:73:1:(b(t,(t+u)/2),b((t+u)/2,u),0);}/*<00>\n
14 _No.          _I_am_          IOCCC          1977          ***/\n
15 #include/***** your father.. --DarthVader */ <time.h>/***** *****/\n
16 int(main)(int (x), char**V){; clock_t(c)= /* */clock();;; for(d=D\n
17 ;m<26;m++,s ++)*s> 63?*d++=m% 7*          16-7 *8,*d++=m/ 7*25,*d++\n
18 =*s-64:0:; if(V[1]) {;;FILE *F =fopen(V[+1], "r");for (d=D,L=N=m\n
19 =0;(*F=fgetc(F))>0

```

Obfuscation techniques

- String obfuscation
- Control flow obfuscation
- Packing
- Bloating
- ...

Why obfuscate strings?

- Strings often have sensitive / important info
- Strings are easy to locate

Unobfuscated example

```
1 char ip[] = "169.254.0.33";
2 char registry_key[] = "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\
  RunOnce";
3
4 int steal_content_from_registry(char *ip, char *registry_key) {
5   printf("%s: %s\n", ip, registry_key);
6   return 1;
7 }
8
9 int main(void) {
10   steal_content_from_registry(ip, registry_key);
11 }
```

Unobfuscated strings are easy to find

pestudio 9.54 - Malware Initial Assessment - www.winitor.com - [z:\windows-malware\main.exe]

file settings about

encod... size (bytes)	locati...	flag (4)	label (375)	group (7)	technique (3)	value (4907)
ascii 12	.idata	-	url-pattern	-	-	169.254.0.33
ascii 49	.idata	-	guid	-	T1060 Registry Run Keys / Startup Folder	SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce
ascii 21	.idata	-	import	synchronizat...	-	DeleteCriticalSection
ascii 20	.idata	-	import	synchronizat...	-	EnterCriticalSection
ascii 25	.idata	-	import	synchronizat...	-	InitializeCriticalSection
ascii 20	.idata	-	import	synchronizat...	-	LeaveCriticalSection
ascii 14	.idata	x	import	memory	T1055 Process Injection	VirtualProtect
ascii 12	.idata	-	import	memory	T1055 Process Injection	VirtualQuery
ascii 6	.idata	-	-	memory	-	malloc
ascii 6	.idata	-	-	memory	-	memcpy
ascii 6	.idata	-	-	memory	-	memset
ascii 5	.idata	-	-	file	-	fputc
ascii 6	.idata	-	-	file	-	fwrite
ascii 11	.idata	-	import	execution	-	TlsGetValue
ascii 5	.idata	-	-	execution	T1497 Sandbox Evasion	Sleep
ascii 27	.idata	-	import	exception	-	SetUnhandledExceptionFilter
ascii 12	.idata	-	import	diagnostic	-	GetLastError
ascii 16	.idata	-	import	-	-	IsDBCSLeadByteEx
ascii 19	.idata	-	import	-	-	MultiByteToWideChar
ascii 19	.idata	-	import	-	-	WideCharToMultiByte
ascii 20	.idata	-	import	-	-	C_specific_handler
ascii 19	.idata	-	import	-	-	Ic_codepage_func

sha256: 34309090D00A2F4F55488CB22A27726F067 cpu: 64 file-type: executable subsystem: console entry-point: 0x00

String obfuscation – encryption

```
1 unsigned char registry_key[] = {161, 189, 180, 166, 165, 179, 160, 183, 174, 191, 155, 145, 128,
2                               157, 129, 157, 148, 134, 174, 165, 155, 156, 150, 157, 133, 129,
3                               174, 177, 135, 128, 128, 151, 156, 134, 164, 151, 128, 129, 155,
4                               157, 156, 174, 160, 135, 156, 189, 156, 145, 151, 0};
5
6 void xor_decrypt(unsigned char key, unsigned char *content) {
7     for (int i = 0; content[i] != 0; ++i) {
8         content[i] = content[i] ^ key;
9     }
10}
11
12 int main(void) {
13     unsigned char key = 242;
14     xor_decrypt(key, registry_key);
15     printf("%s\n", registry_key);
16     // prints: SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce
17 }
```

String obfuscation – encryption

- Strings are just arrays of chars
- Each char can be individually encrypted
- Different encryption techniques possible

String obfuscation – stack strings

```
1 int main(void) {
2     unsigned int ip_0 = 0;
3     unsigned int ip_1 = 0x33332e30;
4     unsigned int ip_2 = 0x2e343532;
5     unsigned int ip_3 = 0x2e393631;
6
7     xor_decrypt(key, registry_key);
8     // int steal_content_from_registry(char *ip, char *registry_key) {};
9     steal_content_from_registry((unsigned char *) &ip_3, registry_key);
10 }
```

String obfuscation – stack strings

- Strings are just arrays of chars
- Arrays are continuous areas of memory
- Stack variables next to each other can be interpreted as an array

Obfuscated strings are hard to find

pestudio 9.54 - Malware Initial Assessment - www.winitor.com - [z:\windows-malware\main-obf.exe]

file settings about

encodin...	size (...)	locati...	flag (4)	label (375)	group (7)	technique (2)	value (4938)
z:\windows-malware\main-obf.exe							
indicators (sections > count)							
footprints (count > 25) *							
virustotal (offline)							
dos-header (size > 64 bytes)							
dos-stub (size > 64 bytes)							
rich-header (n/a)							
file-header (sections-count)							
optional-header (subsystem > console)							
directories (count > 6)							
sections (characteristics > virtual)							
libraries (count > 2)							
imports (flag > 47) *							
exports (n/a)							
thread-local-storage (count > 2)							
.NET (n/a)							
resources (signature > manifest)							
strings (count > 4938)							
debug (n/a)							
manifest (level > asInvoker)							
version (n/a)							
certificate (n/a)							
overlay (signature > MinGW)							
sha256: 9D5C5259A3E7F5C6A40123CD5F5120E414 cpu: 64 file-type: executable subsystem: console entry-point: 0x00							
encodin...	size (...)	locati...	flag (4)	label (375)	group (7)	technique (2)	value (4938)
ascii 21	.idata	-	x import	synchronization	-	T1055 Process Inj...	DeleteCriticalSection
ascii 20	.idata	-	x import	synchronization	-	T1055 Process Inj...	EnterCriticalSection
ascii 25	.idata	-	x import	synchronization	-	T1055 Process Inj...	InitializeCriticalSection
ascii 20	.idata	-	x import	synchronization	-	T1055 Process Inj...	LeaveCriticalSection
ascii 14	.idata	x	x import	memory	T1055 Process Inj...	VirtualProtect	
ascii 12	.idata	-	x import	memory	T1055 Process Inj...	VirtualQuery	
ascii 6	.idata	-	-	memory	-	-	malloc
ascii 6	.idata	-	-	memory	-	-	memcpy
ascii 6	.idata	-	-	memory	-	-	memset
ascii 5	.idata	-	-	file	-	-	fputc
ascii 6	.idata	-	-	file	-	-	fwrite
ascii 11	.idata	-	x import	execution	-	-	TlsGetValue
ascii 5	.idata	-	-	execution	T1497 Sandbox E...	Sleep	
ascii 27	.idata	-	x import	exception	-	-	SetUnhandledExceptionFilter
ascii 12	.idata	-	x import	diagnostic	-	-	GetLastError
ascii 16	.idata	-	x import	-	-	-	IsDBCSLeadByteEx
ascii 19	.idata	-	x import	-	-	-	MultiByteToWideChar
ascii 19	.idata	-	x import	-	-	-	WideCharToMultiByte
ascii 20	.idata	-	x import	-	-	-	C_specific_handler
ascii 19	.idata	-	x import	-	-	-	Ic_codepage_func
ascii 18	.idata	-	x import	-	-	-	mb_cur_max_func
ascii 13	.idata	-	x import	-	-	-	getmainargs
ascii 9	.idata	-	x import	-	-	-	initenv

Deobfuscation

- Static analysis
- Emulation
- Debugger
- ...

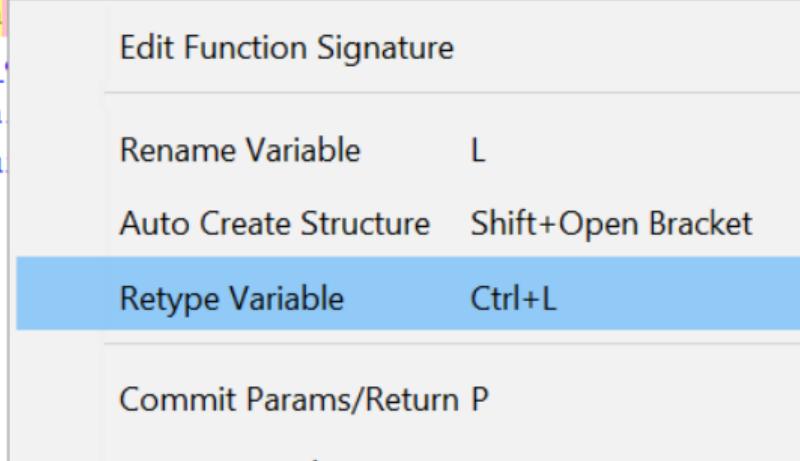
Deobfuscating stack strings via retyping

```
MOV     dword ptr [RBP + local_c],0x0
        ...
MOV     dword ptr [RBP + local_10],0x33332e30
        ...
MOV     dword ptr [RBP + local_14],0x2e343532
        ...
MOV     dword ptr [RBP + local_18],0x2e393631
```

```
10    __main();
11    local_c = 0;
12    local_10 = 0x33332e30;
13    local_14 = 0x2e343532;
14    local_18 = 0x2e393631;
15    xor_decrypt((byte)key,0x140008000);
16    steal_content_from_registry(&local_18,&DAT_140008000);
17    return 0;
18 }
19 }
```

Deobfuscating stack strings via retyping

```
9
10    __main();
11    local_c = 0;
12    local_10 = 0x33332e30;
13    local_14 = 0x2e343532;
14    loca
15    xor_
16    stea
17    retu
18 }
19 }
```

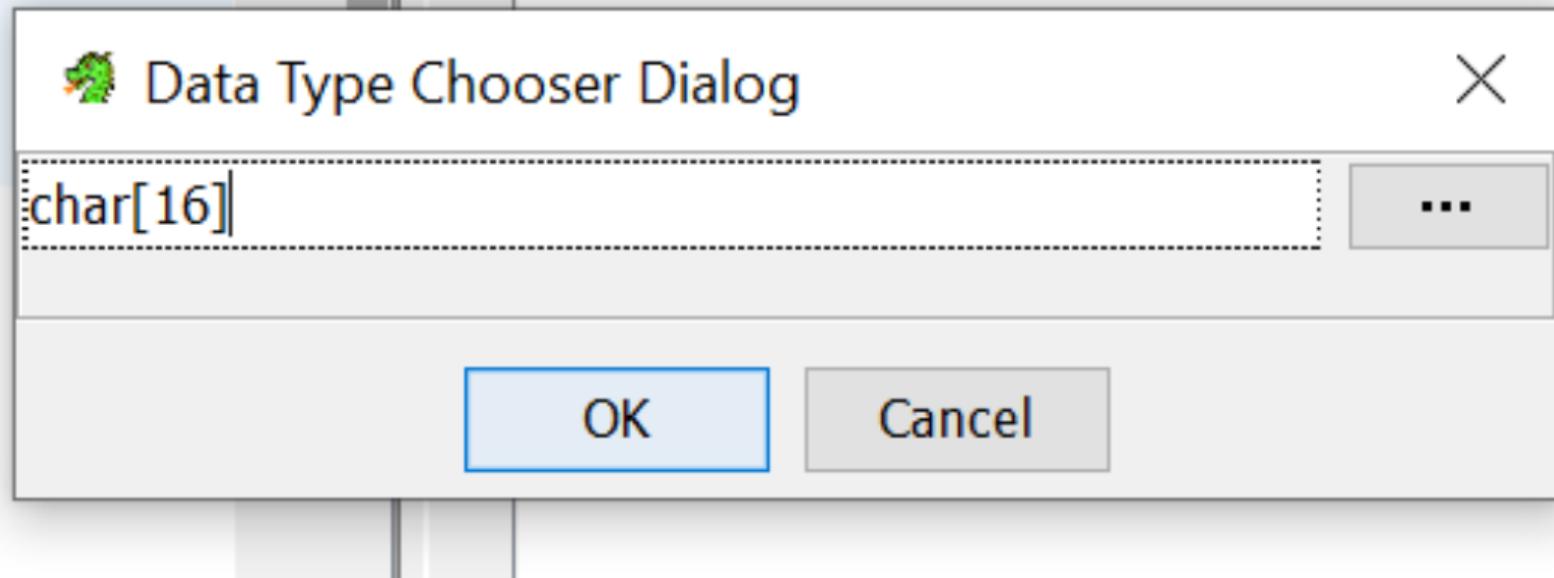


A context menu is displayed over the variable declaration 'local_10'. The menu items are:

- Edit Function Signature
- Rename Variable L
- Auto Create Structure Shift+Open Bracket
- Retype Variable Ctrl+L**
- Commit Params/Return P

Deobfuscating stack strings via retyping

531 17 return 0;



Deobfuscating stack strings via retyping

```
7  __main();
8  local_18[12] = '\0';
9  local_18[13] = '\0';
10 local_18[14] = '\0';
11 local_18[15] = '\0';
12 local_18[8] = '0';
13 local_18[9] = '.';
14 local_18[10] = '3';
15 local_18[11] = '3';
16 local_18[4] = '2';
17 local_18[5] = '5';
18 local_18[6] = '4';
19 local_18[7] = '.';
20 local_18[0] = '1';
21 local_18[1] = '6';
22 local_18[2] = '9';
23 local_18[3] = '.';
24 xor_decrypt((byte)key,0x140008000);
25 steal_content_from_registry(local_18,&DAT_140008000);
26 return 0;
27 }
```

Deobfuscating encrypted strings via emulation

Original C code from our example malware:

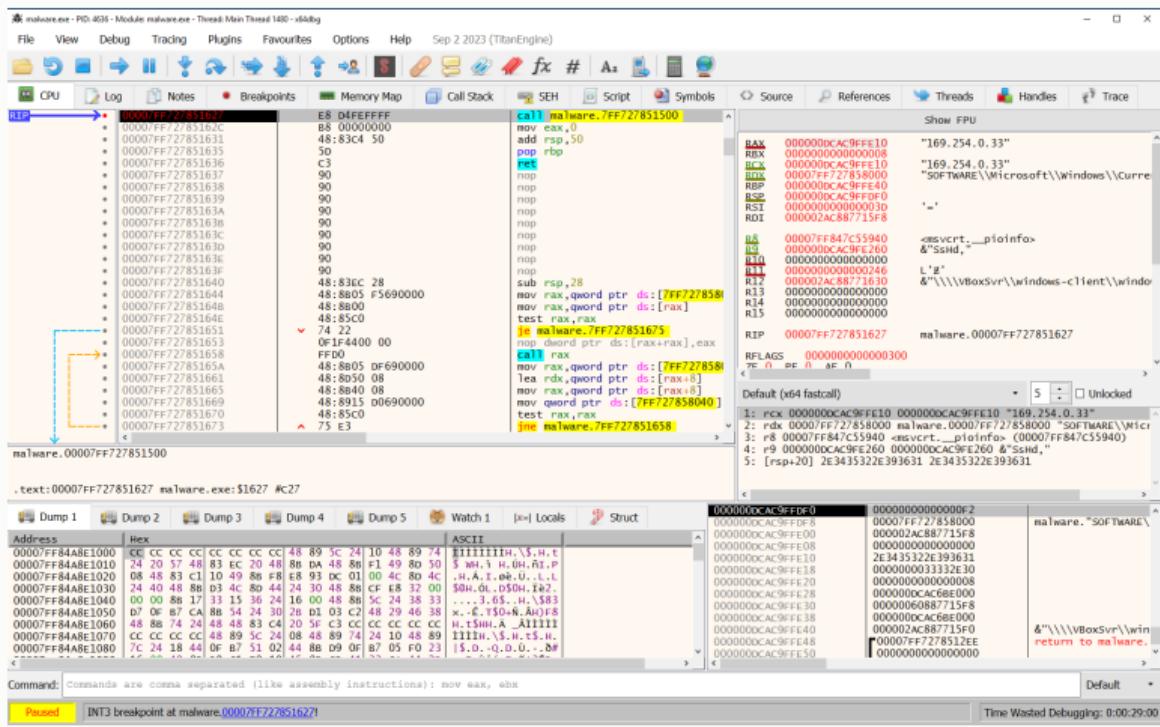
```
1 void xor_decrypt(byte param_1, char *param_2) {
2     for (int local_c = 0; param_2[local_c] != '\0'; local_c = local_c + 1) {
3         param_2[local_c] = param_2[local_c] ^ param_1;
4     }
5 }
```

Deobfuscating encrypted strings via emulation

Re-implemented python version:

```
1 def xor_decrypt(key: int, content: list[int]) -> None:
2     i = 0
3     while content[i] != 0:
4         content[i] = content[i] ^ key
5         i += 1
6
7 secret = [ 161, 189, 180, 166, 165, 179, 160, 183, 174, 191, 155, 145, 128,
8           157, 129, 157, 148, 134, 174, 165, 155, 156, 150, 157, 133, 129,
9           174, 177, 135, 128, 128, 151, 156, 134, 164, 151, 128, 129, 155,
10          157, 156, 174, 160, 135, 156, 189, 156, 145, 151, 0 ]
11 key = 242
12 xor_decrypt(key, secret)
13 print(''.join(chr(c) for c in secret))
14 # >>> SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce
```

Deobfuscating with x64dbg



Deobfuscating with x64dbg

x64dbg interface showing the deobfuscation process for malware.exe.

Code View: Displays assembly code for address `malware.00007FF727851500`. The code is heavily obfuscated, containing multiple `nop` instructions and complex memory operations. A yellow box highlights the `mov rax, qword ptr [rax+rax]` instruction at `FF727851500`.

Registers: Shows CPU register values. The `RAX` register contains the value `0000000000000000`, and the `RSI` register contains `000007FF727851627`.

Call Arguments: Shows the stack frame for the current function. It includes parameters for `RDI` (`0000000000000000`), `RBP` (`000007FF727851627`), and `R9` (`0E3435322E393631`).

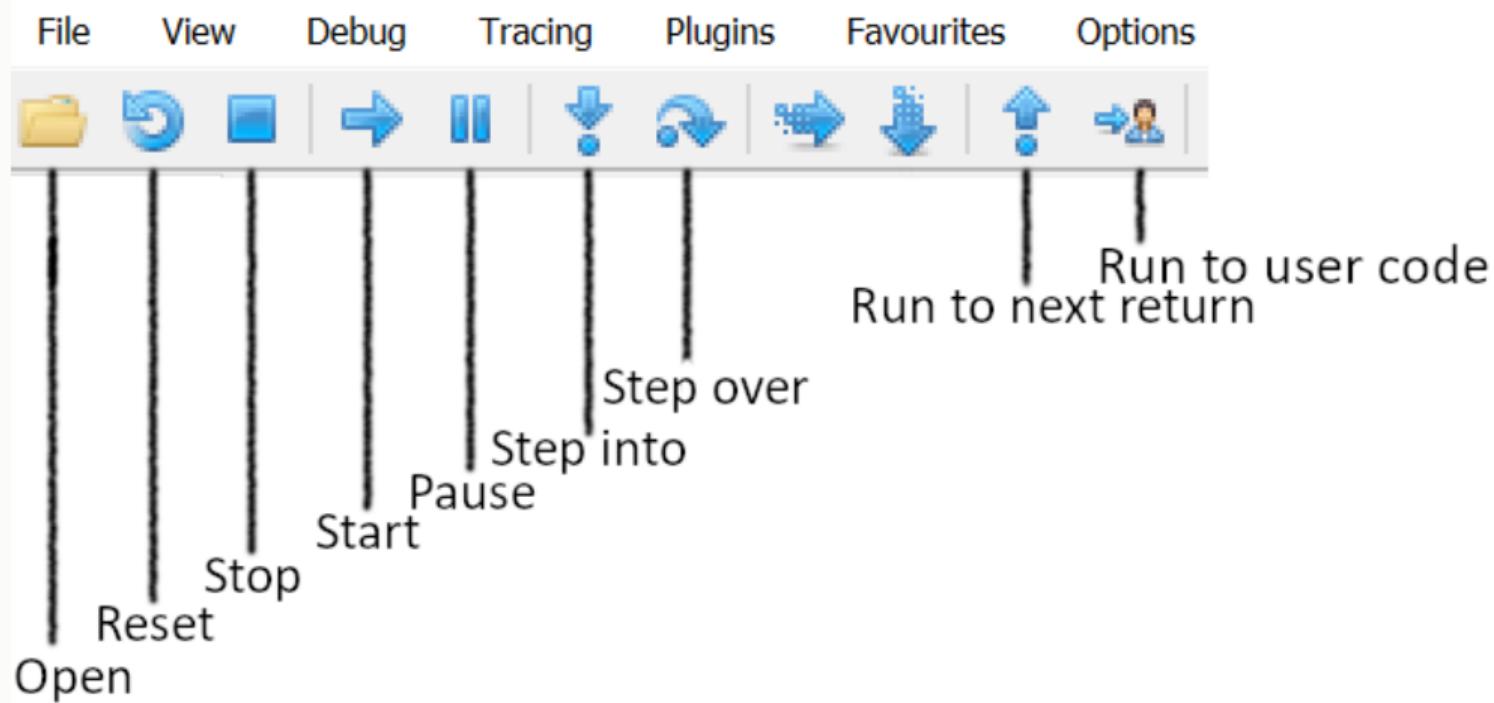
Memory Dump: Shows a dump of memory starting at address `00007FF72784BE1000`. The dump includes columns for Address, Hex, ASCII, and Struct. The ASCII column shows various control characters and some readable text like "11111111\A\B\1", "S\H\A\T\0\H\B\1", and "SOH\0L\SOH\1Z\2".

Stack: Shows the current state of the stack. The top of the stack contains `0000000000000000`.

Command: The command entered is `mov eax, ebx`.

Status: The status bar indicates an **INT3 breakpoint** at address `00007FF727851627`. The total time wasted is 0:00:29.00.

Deobfuscating with x64dbg



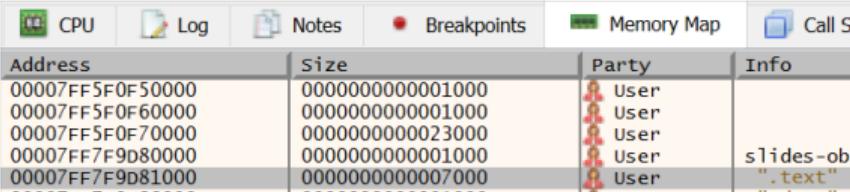
Deobfuscating with x64dbg

```
1 unsigned char registry_key[] = {161, 189, ...};  
2  
3 int main(void) {  
4     unsigned int ip_0 = 0;  
5     unsigned int ip_1 = 0x33332e30;  
6     unsigned int ip_2 = 0x2e343532;  
7     unsigned int ip_3 = 0x2e393631;  
8  
9     xor_decrypt(key, registry_key);  
10    steal_content_from_registry((unsigned char *) &ip_3, registry_key);  
11 }
```

Deobfuscating with x64dbg

- Run until we expect to find clear text data
- Function call at line 10 seems like a good candidate
- Use x64dbg's registers view to extract strings
- How do we know the function call position within the binary?

Find specific address in x64dbg

1. 
2. 
3. 

Deobfuscating with x64dbg

x64dbg interface showing assembly code and registers during deobfuscation.

Registers:

Register	Value	Description
RAX	00000033595FFC10	"169.254.0.33"
RBX	0000000000000008	"169.254.0.33"
RCX	00000033595FFC10	"169.254.0.33"
RDX	00007FF7F9D88000	"SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\RunOnce"
RBP	00000033595FFC20	
RSP	00000033595FFBF0	
RSI	0000000000000003F	
RDI	00000150E3d915f8	

Stack Dump:

```

00007FF7F9D8155A C745 F0 3136 mov dword ptr ss:[rbp-10],2E393631
00007FF7F9D81561 8805 CD6A000 mov eax,dword ptr ds:[7FF7F9D88034]
0FB6C0 movzx eax,al
48:8015 8F6A lea rdx,qword ptr ds:[7FF7F9D88000]
mov ecx,eax
89C1
E8 2CFFFFFF call slides-obfuscated.7FF7F9D814A4
48:8045 F0 lea rax,qword ptr ss:[rbp-10]
48:8015 7D6A lea rdx,qword ptr ds:[7FF7F9D88000]
48:89C1 mov rcx,rax
E8 75FFFFFF call slides-obfuscated.7FF7F9D81500
    
```

Deobfuscating with x64dbg

- Last 2 bytes of address are usually static
- Find them e.g. in ghidra
- Look up start of .text section in memory map on x64dbg
- Set breakpoint at desired offset
- Function arguments are found in the registers section

Summary

- Obfuscation makes analysis hard
- Static analysis can help for simple things
- Dynamic analysis can be more powerful, but also more challenging
- There is no silver bullet

Questions?

Anti-Debugging

-  Motivation
-  Getting Infected
 - PDF Analysis
-  PE Analysis
 - Obfuscation
 - Anti-Debugging
 - Anti-Sandboxing
-  Outlook

About Anti-Debugging

- Makes dynamic analysis harder
- Various techniques
 - Windows' Win32 API
 - Windows internal flags
 - Many, many side effects (see <https://anti-debug.checkpoint.com>)

- Built-In Windows API
- Function IsDebuggerPresent() from debugapi.h
- Intended to enable applications to provide additional information

```
1 #include <windows.h>
2
3 int main(void) {
4     if (IsDebuggerPresent()) {
5         do_some_evasive_stuff_like_exit();
6     } else {
7         do_evil_stuff();
8     }
9 }
```

Anti-Debug — Process Environment Block

- Contains some process context information
- For example the BeingDebugged flag
- Internal structure → no stable API

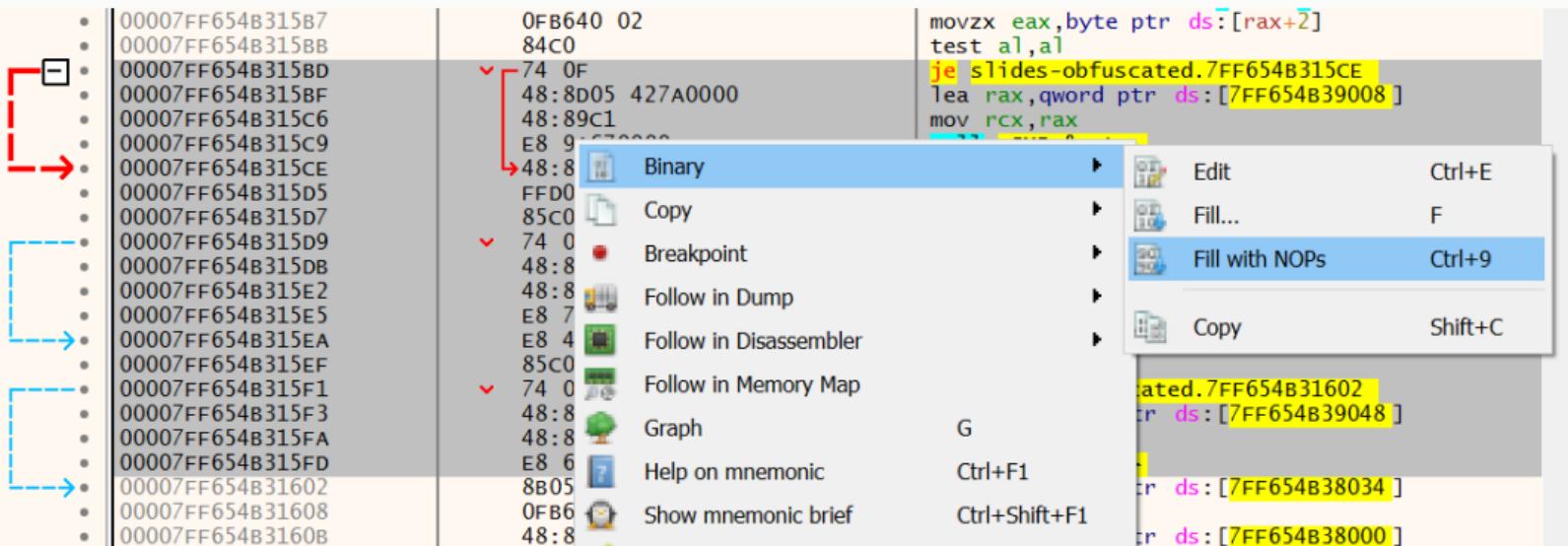
Anti-Debug — PEB

```
1 #include <windows.h>
2
3 int main(void) {
4     // decompilation shows something like `*(unaff_GS_OFFSET + 0x60)`
5     PPEB pPeb = (PPEB)__readgsqword(0x60);
6     if (pPeb->BeingDebugged) {
7         do_some_evasive_stuff_like_exit();
8     } else {
9         do_evil_stuff();
10    }
11 }
```

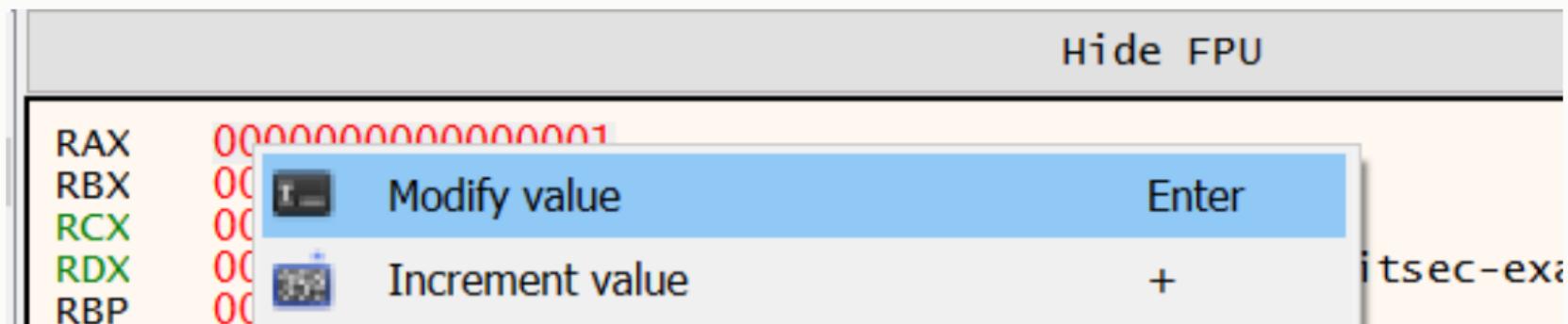
Anti-Debug Circumvention

- Patching
- Modify Return Values
- ScyllaHide for x64DBG
- Hook API calls

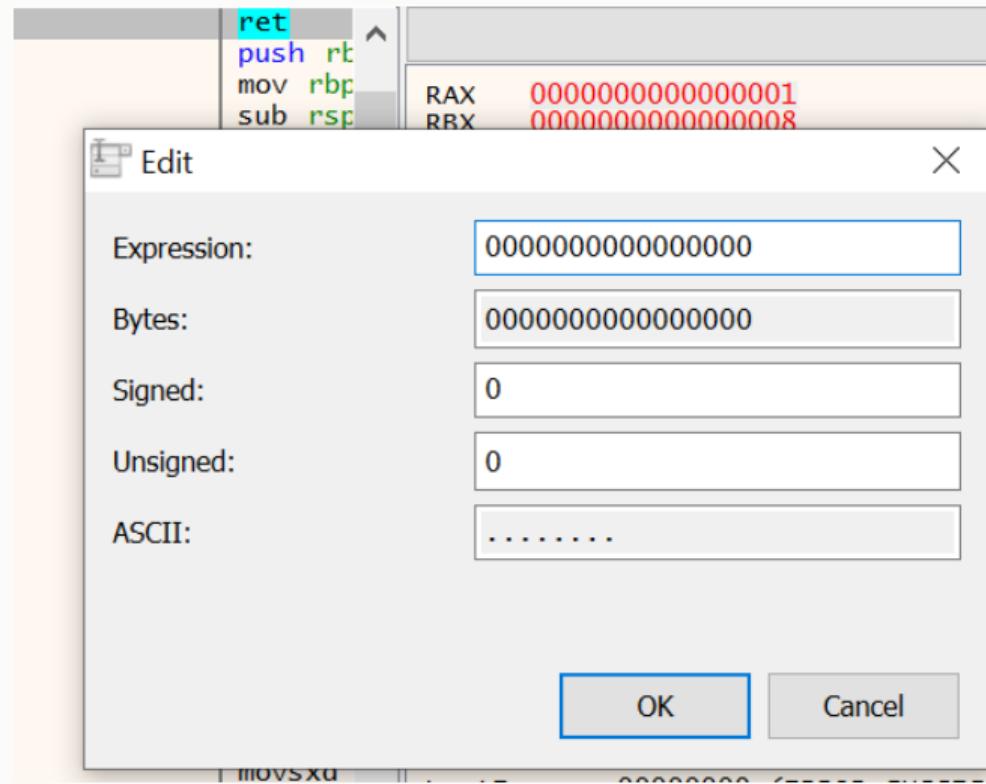
Anti-Debug Circumvention — Patching



Anti-Debug Circumvention — Modify Return Values

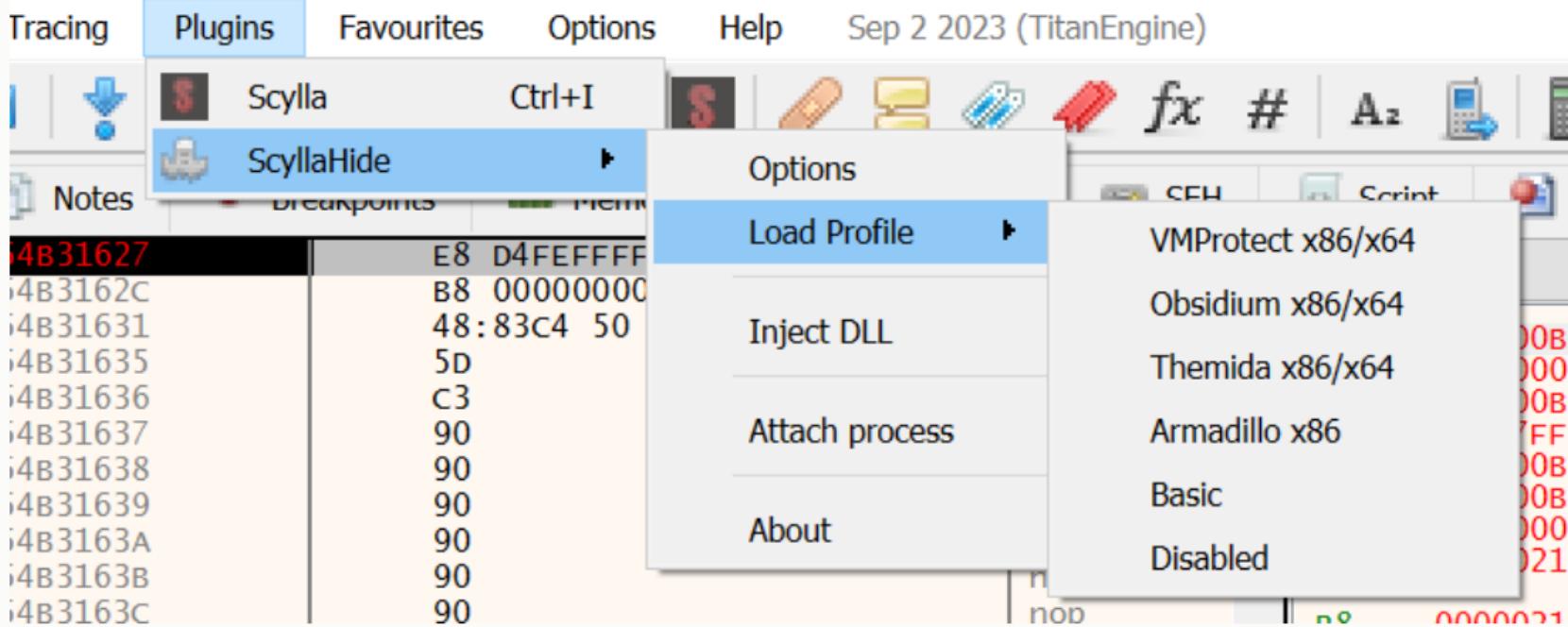


Anti-Debug Circumvention — Modify Return Values



Anti-Debug Circumvention — ScyllaHide

PID: 6132 - Module: slides-obfuscated.exe - Thread: Main Thread 1636 - x64dbg



The screenshot shows the x64dbg debugger interface. The top menu bar includes Tracing, Plugins (which is currently selected), Favourites, Options, Help, and a date/time stamp (Sep 2 2023 (TitanEngine)). Below the menu bar is a toolbar with various icons. The main window displays assembly code and memory dump sections. A context menu is open over the assembly section, listing options: Load Profile, Inject DLL, Attach process, and About. The "Load Profile" option is highlighted. To the right of the assembly section, a list of anti-debug profiles is shown, each with a preview icon and name: VMProtect x86/x64, Obsidium x86/x64, Themida x86/x64, Armadillo x86, Basic, and Disabled. The "ScyllaHide" profile is currently selected, indicated by a blue highlight.

Tracing Plugins Favourites Options Help Sep 2 2023 (TitanEngine)

Scylla Hide Ctrl+I Options

Notes Breakpoints Memory

i4B31627 E8 D4FFFFF
i4B3162C B8 00000000
i4B31631 48:83c4 50
i4B31635 5D
i4B31636 C3
i4B31637 90
i4B31638 90
i4B31639 90
i4B3163A 90
i4B3163B 90
i4B3163C 90

Load Profile

VMProtect x86/x64

Obsidium x86/x64

Themida x86/x64

Armadillo x86

Basic

Disabled

Summary

- Anti-Debug techniques
 - Use / abuse various mechanisms to detect debugger presence
 - Lead to alternative code paths
- Anti-Debug circumvention
 - Jump over anti-debug routines
 - Change return values of detection routines
 - Use existing tools

Questions?

Anti-Sandboxing

-  Motivation
-  Getting Infected
 - PDF Analysis
-  PE Analysis
 - Obfuscation
 - Anti-Debugging
 - Anti-Sandboxing
-  Outlook

Anti-Sandbox Techniques

- File system
- Processor instructions
- Hardware detection
- Network
- Behaviour
- Guest Additions
- ... (<https://evasions.checkpoint.com>)

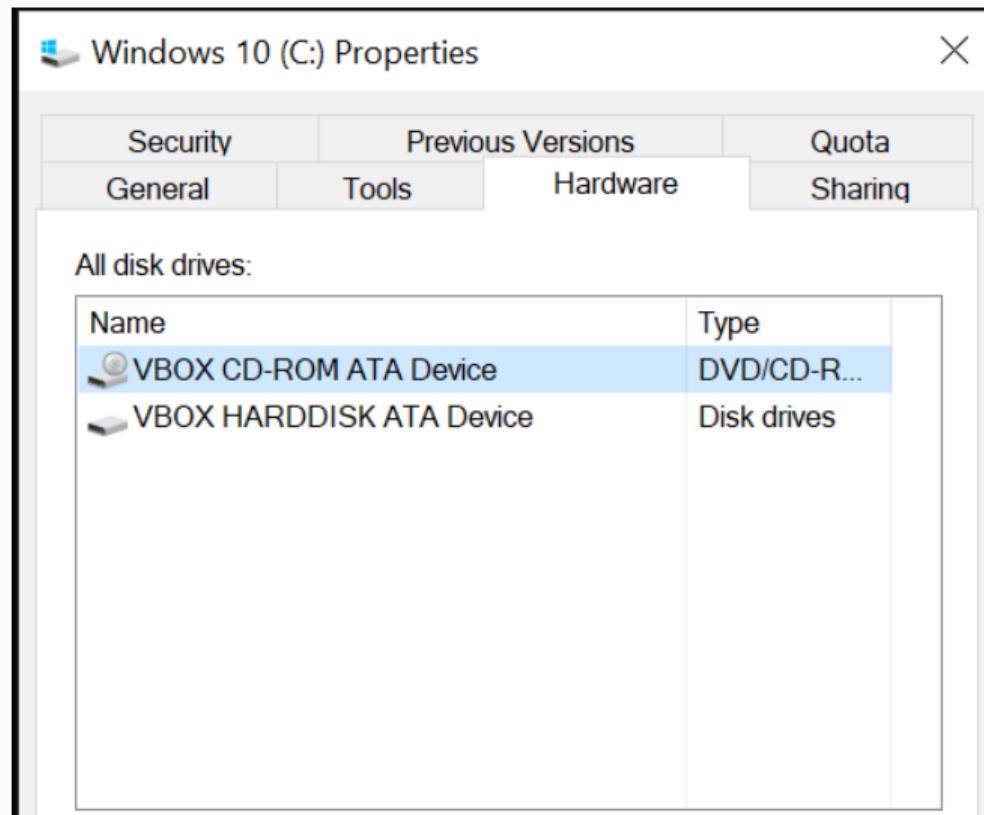
Anti-Sandbox — File system artifacts

- Files specific to Virtual Machine
 - System32\drivers\VBoxMouse.sys
 - System32\drivers\VBoxVideo.sys
 - ...
- Binary path contains keywords (“sample”, “malware”, ...)
 - ...

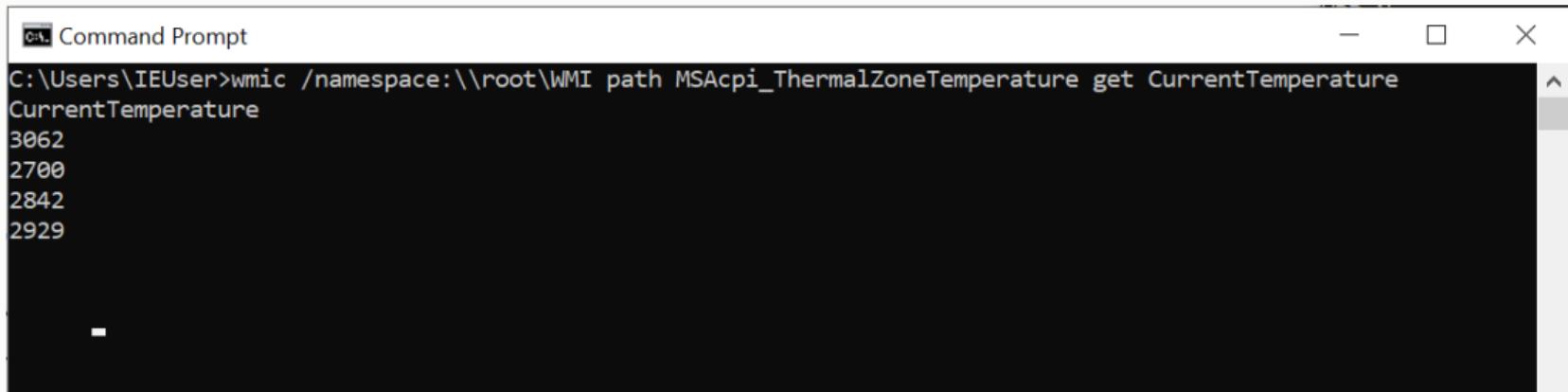
- CPUID instruction
 - Returns CPU information like the vendor name
 - In sandboxed environments usually a sandbox-specific string
- Exotic instructions
 - Sandboxes often don't implement all CPU instructions
- Undocumented instructions
- ...

- Device Names
 - In sandboxed environments usually sandbox-specific
- CPU Temperature Information
 - Sensors often not available to sandbox
- Audio device availability
- ...

Anti-Sandbox — Device Names



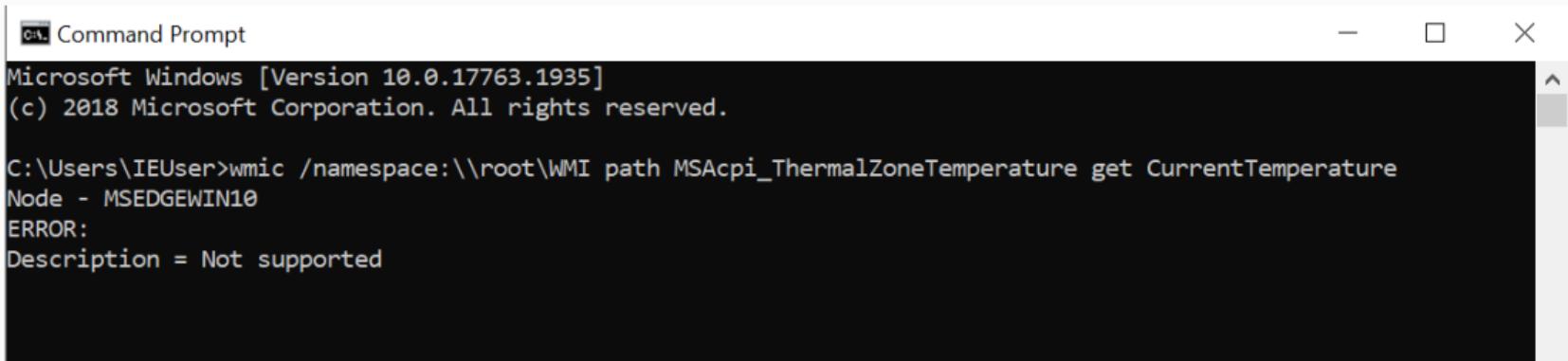
CPU Temperature Information on Bare Metal



A screenshot of a Windows Command Prompt window titled "Command Prompt". The window contains the following text:

```
C:\Users\IEUser>wmic /namespace:\\root\WMI path MSAcpi_ThermalZoneTemperature get CurrentTemperature
CurrentTemperature
3062
2700
2842
2929
```

CPU Temperature Information in a Sandbox



Command Prompt

```
Microsoft Windows [Version 10.0.17763.1935]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\IEUser>wmic /namespace:\\root\WMI path MSAcpi_ThermalZoneTemperature get CurrentTemperature
Node - MSEDGEWIN10
ERROR:
Description = Not supported
```

Anti-Sandbox – Mitigations

- Existing tools
 - <https://github.com/nsmfoo/antivmdetection>
 - <https://github.com/hfiref0x/VBoxHardenedLoader>
- Painfully do it yourself
- Test your hardening with other tools
 - <https://github.com/LordNoteworthy/al-khaser>
 - <https://github.com/a0rtega/pafish>

Summary

- There are *many* things giving away a sandboxed environment
- Some are of technical nature due to the environment
- Some are due to the absence of a real user
- Some are due to the habits of researchers
- Only mitigation is to know them all (or use tools from authors that do)

Questions?

Outlook

-  Motivation
-  Getting Infected
 -  PDF Analysis
-  PE Analysis
 -  Obfuscation
 -  Anti-Debugging
 -  Anti-Sandboxing
-  Outlook



Lightning
Surveys 

Outlook

- Infection chains
- Payloads
- Process Injections
- Persistence
- Much more detail
- Real malware
- Practical Network Analysis



Lightning
Surveys 

Timo Pohl

University of Bonn | Institute of Computer Science 4

pohl@cs.uni-bonn.de

