Segurança de Sistemas e dados (MSI 2021/2022)

Aula 9

Rolando Martins
DCC – FCUP
Slides Adaptados do Prof. Manuel Eduardo Correia

Can we ever Trust Software?

Trusting Software

* Can you ever trust software?
* Consider the following thought experiment
* Suppose C compiler has a virus
  * When compiling login program, virus creates backdoor (account with known password)
  * When recompiling the C compiler, virus incorporates itself into new C compiler
* Difficult to get rid of this virus!

Trusting Software

BEHIND THE CODE

Malware In GitHub Repositories

JULY 15, 2021 in Behind the Code

Overview

It is unsurprising to find malware hosted on GitHub. GitHub, being a free website specifically geared towards hosting and deploying code for millions of people and organizations, which makes it an ideal location for malicious actors to hide their own code. Whether pulling from their own repositories or pulling from the handy collections of malware analysts, bad actors have a handy location for their malware to reside.

A recent investigation uncovered two previously unexpected locations where malware could be found:

- The repository description
- Easily parsed Markdown files

A crafty attacker can easily use these innocuous locations to successfully hide and deploy a payload from GitHub than using traditional file-based methods. As such, malware analysts and researchers need to be on the lookout for additional non-traditional retrieval methods from GitHub as well as any manipulation of the retrieved content.

https://www.sitelock.com/blog/malware-in-github-repositories/

ars TECHNICA

BIZ & IT TECH SCIENCE POLICY CARS GAMING & CULTURE STORE FORUM

SHARE AND SHARE ALIKE —

Software downloaded 30,000 times from PyPI ransacked developers' machines

Expect to see more of these "Frankenstein" malware packages, researchers warn.

DAN GOODIN - 7/30/2021, 12:54 PM

Enlarge

121

Open source packages downloaded an estimated 30,000 times from the PyPI open source repository contained malicious code that surreptitiously stole credit card data and login credentials and injected malicious code on infected machines, researchers said on Thursday.

In a post, researchers Andrey Polkovnichenko, Omer Kaspi, and Shachar Menashe of devops software vendor JFrog said they recently found eight packages in PyPI that carried out a range of malicious activity. Based on searches on https://pepy.tech, a site that provides download stats for Python packages, the researchers estimate the malicious packages were downloaded about 30,000 times.

Systemic threat

The discovery is the latest in a long line of attacks in recent years that abuse the receptivity of open source repositories, which millions of software developers rely on daily. Despite

FURTHER READING
Ahoy, there's malice in your repos—PyPI is the latest to be abused

https://arstechnica.com/gadgets/2021/07/malicious-pypi-packages-caught-stealing-developer-data-and-injecting-code/

Trusting Software

* Suppose you notice something is wrong
* So you start over from scratch
* First, you recompile the C compiler
* Then you recompile the OS
  * Including login program…
  * You have not gotten rid of the problem!
* In the real world
  * Attackers try to hide viruses in virus scanner
  * Imagine damage that would be done by attack on virus signature updates

Trusting Software

* Real Word Case (undisclosed Portuguese Institution/company)
* First step, organization hacked
* How? Mismanaged VPN by a third party/subcontractor
* Followed by an attempt of data extraction…
* Using malware.
* **How can you recover from this?**

Software Reverse Engineering (SRE)

SRE

* **Software Reverse Engineering**
  * Also known as Reverse Code Engineering (RCE)
  * Or simply "reversing"
* Can be used for **good**...
  * Understand malware
  * Understand legacy code
* ... or **not-so-good**
  * Remove usage restrictions from software
  * Find and exploit flaws in software
  * Cheat at games, etc.

SRE

* We assume that
  * Reverse engineer is an attacker
  * Attacker only has exe (no source code)

* Attacker might want to
  * Understand the software
  * Modify the software

SRE Tools

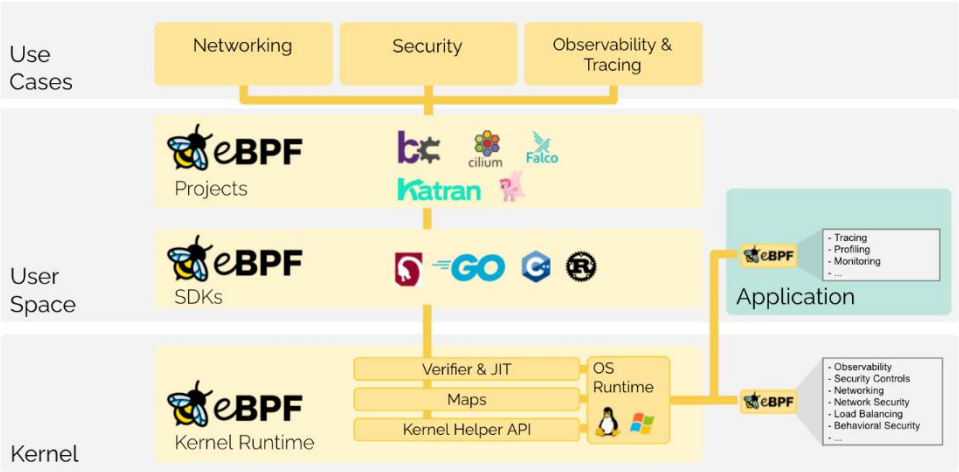
* Disassembler
  * Converts exe to assembly — as best it can
  * Cannot always disassemble correctly
  * In general, it is not possible to assemble disassembly into working exe
* Debugger
  * Must step through code to completely understand it
  * Labor intensive — lack of automated tools
* Hex Editor
  * To **patch** (make changes to) exe file

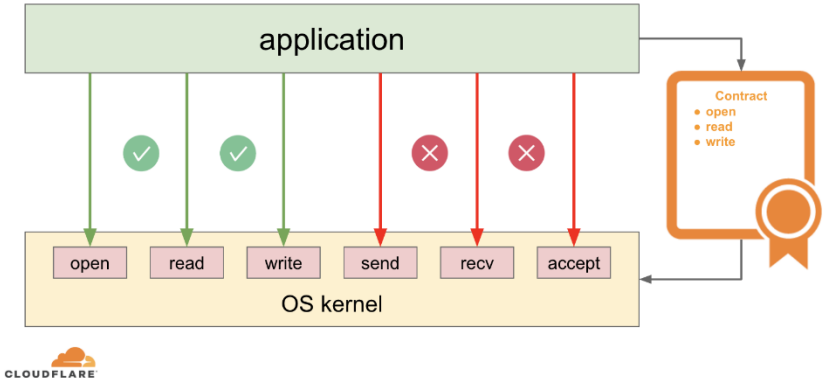
SRE Tools

* **IDA Pro** is the top-rated disassembler
  * Cost is a few hundred dollars
  * Converts binary to assembly (as best it can)
* **SoftICE** is "alpha and omega" of debuggers
  * Cost is in the $1000's
  * Kernel mode debugger
  * Can debug anything, even the OS
* **OllyDbg** is a high quality shareware debugger
  * Includes a good disassembler
* **Hex editor** — to view/modify bits of exe
  * UltraEdit is good — freeware
  * HIEW — useful for patching exe
* Strace, GDD, …

Quasi-SRE : tracing and sanboxing

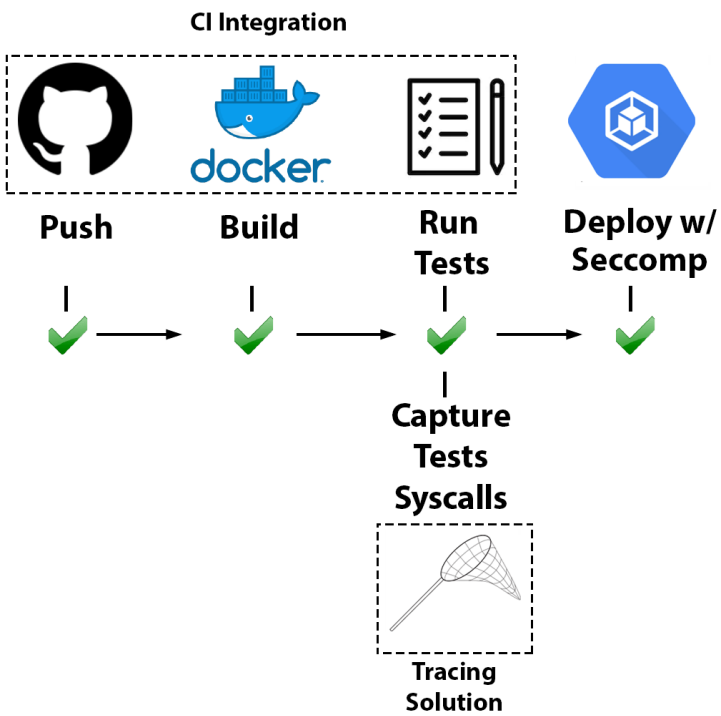
* **And more recently, eBPF and Seccomp:**

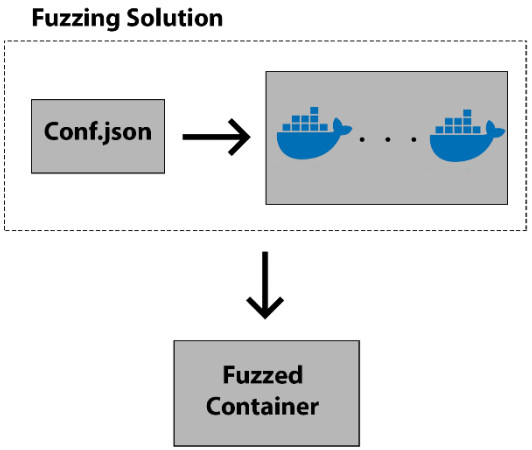


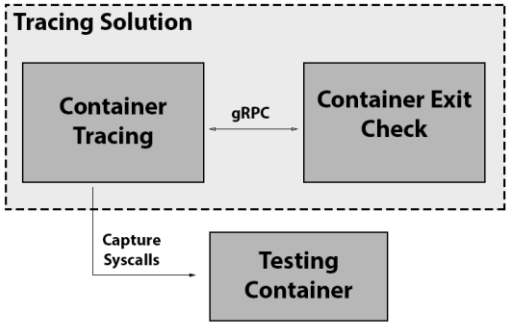


Quasi-SRE :

"Container Hardening Through Automated Seccomp Profiling", WOC@Middleware'20







Why is a Debugger Needed?

* Disassembler gives **static** results
  * Good overview of program logic
  * But need to "mentally execute" program
  * Difficult to jump to specific place in the code
* Debugger is **dynamic**
  * Can set break points
  * Can treat complex code as "black box"
  * Not all code disassembles correctly
* Disassembler **and** debugger both required for any serious SRE task

SRE Necessary Skills

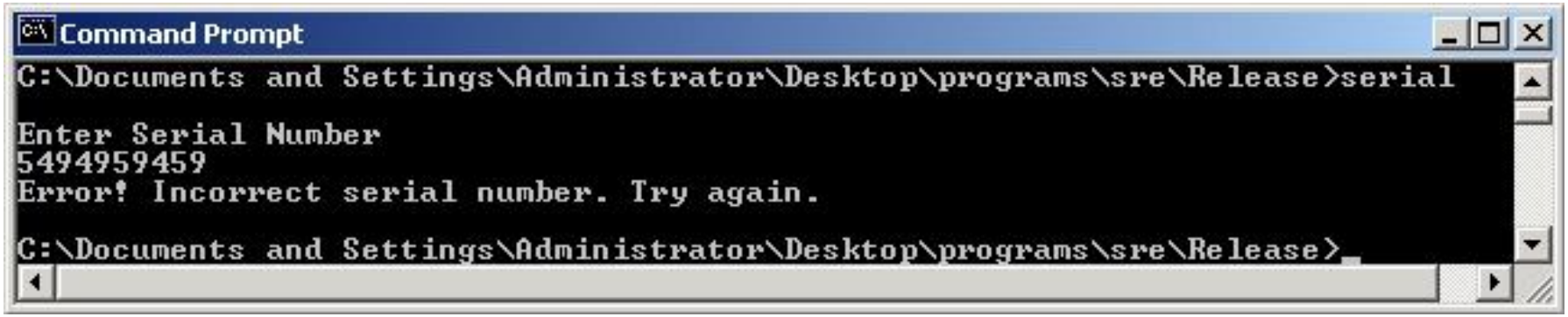
* Working knowledge of target assembly code
* Experience with the tools
  * IDA Pro — sophisticated and complex
  * SoftICE — large two-volume users manual
* Knowledge executable format:
  * **Portable Executable** (PE) file for Windows
  * **ELF** for UNIX/Linux.
* Boundless patience and optimism
* SRE is tedious and labor-intensive process!

SRE Example

* Consider simple example
* This example only requires disassembler (IDA Pro) and hex editor
  * Trudy disassembles to understand code
  * Trudy also wants to patch the code
* For most real-world code, also need a debugger (**SoftICE** or **OllyDbg**)

SRE Example

* Program requires serial number
* But Trudy doesn't know the serial number!



- Can Trudy find the serial number?

SRE Example

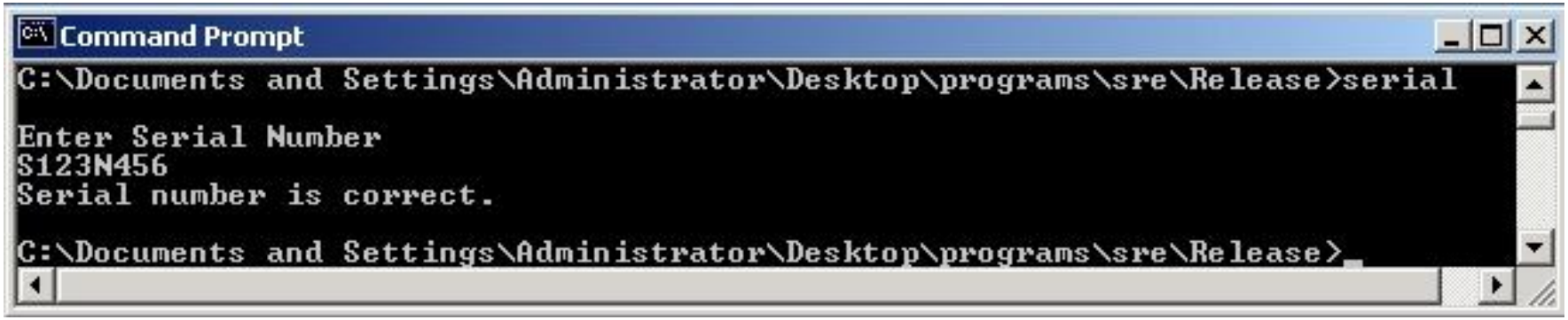
* IDA Pro disassembly

```
.text:00401003    push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008    call    sub_4010AF
.text:0040100D    lea     eax, [esp+18h+var_14]
.text:00401011    push    eax
.text:00401012    push    offset aS        ; "%s"
.text:00401017    call    sub_401098
.text:0040101C    push    8
.text:0040101E    lea     ecx, [esp+24h+var_14]
.text:00401022    push    offset aS123n456 ; "S123N456"
.text:00401027    push    ecx
.text:00401028    call    sub_401060
.text:0040102D    add     esp, 18h
.text:00401030    test    eax, eax
.text:00401032    jz      short loc_401045
.text:00401034    push    offset aErrorIncorrect ; "Error! Incorrect serial number.
.text:00401039    call    sub_4010AF
```

- Looks like serial number is S123N456

SRE Example

* Try the serial number S123N456



- It works!
- Can Trudy do better?

SRE Example

* Again, IDA Pro disassembly

```
.text:00401003    push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008    call    sub_4010AF
.text:0040100D    lea     eax, [esp+18h+var_14]
.text:00401011    push    eax
.text:00401012    push    offset aS       ; "%s"
.text:00401017    call    sub_401098
.text:0040101C    push    8
.text:0040101E    lea     ecx, [esp+24h+var_14]
.text:00401022    push    offset aS123n456 ; "S123N456"
.text:00401027    push    ecx
.text:00401028    call    sub_401060
.text:0040102D    add     esp, 18h
.text:00401030    test    eax, eax
.text:00401032    jz      short loc_401045
.text:00401034    push    offset aErrorIncorrect ; "Error! Incorrect serial number.
.text:00401039    call    sub_4010AF
```

- And hex view…

```
.text:00401010  04 50 68 84 80 40 00 E8-7C 00 00 00 6A 08 8D 4C
.text:00401020  24 10 68 78 80 40 00 51-E8 33 00 00 00 83 C4 18
.text:00401030  85 C0 74 11 68 4C 80 40-00 E8 71 00 00 00 83 C4
.text:00401040  04 83 C4 14 C3 68 30 80-40 00 E8 60 00 00 00 83
```

SRE Example

```
.text:00401003    push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008    call    sub_4010AF
.text:0040100D    lea     eax, [esp+18h+var_14]
.text:00401011    push    eax
.text:00401012    push    offset aS       ; "%s"
.text:00401017    call    sub_401098
.text:0040101C    push    8
.text:0040101E    lea     ecx, [esp+24h+var_14]
.text:00401022    push    offset aS123n456 ; "S123N456"
.text:00401027    push    ecx
.text:00401028    call    sub_401060
.text:0040102D    add     esp, 18h
.text:00401030    test    eax, eax
.text:00401032    jz      short loc_401045
.text:00401034    push    offset aErrorIncorrect ; "Error! Incorrect serial number.
.text:00401039    call    sub_4010AF
```

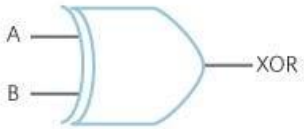
- test eax,eax gives AND of eax with itself
  - Result is 0 only if eax is 0
  - If test returns 0, then jz is true
- Trudy wants jz to always be true!
- Can Trudy patch exe so that jz always true?

SRE Example

- Can Trudy patch exe so that jz always true?

```
.text:00401003    push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008    call    sub_4010AF
.text:0040100D    lea     eax, [esp+18h+var_14]
.text:00401011    push    eax
.text:00401012    push    offset aS        ; "%s"
.text:00401017    call    sub_401098
.text:0040101C    push    8
.text:0040101E    lea     ecx, [esp+24h+var_14]
.text:00401022    push    offset aS123n456 ; "S123N456"
.text:00401027    push    ecx
.text:00401028    call    sub_401060
.text:0040102D    add     esp, 18h
.text:00401030    xor     eax, eax
.text:00401032    jz      short loc_401045  ← jz always true!!!
.text:00401034    push    offset aErrorIncorrect ; "Error! Incorrect serial number.
.text:00401039    call    sub_4010AF
```

$X = A \oplus B$



A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Assembly	Hex
test eax,eax	85 C0 …
xor eax,eax	33 C0 …

SRE Example

* Edit serial.exe with hex editor

serial.exe

```
00001010h: 04 50 68 84 80 40 00 E8 7C 00 00 00 6A 08 8D 4C
00001020h: 24 10 68 78 80 40 00 51 E8 33 00 00 00 83 C4 18
00001030h: 85 C0 74 11 68 4C 80 40 00 E8 71 00 00 00 83 C4
00001040h: 04 83 C4 14 C3 68 30 80 40 00 E8 60 00 00 00 83
00001050h: C4 04 83 C4 14 C3 90 90 90 90 90 90 90 90 90 90
```

serialPatch.exe

```
00001010h: 04 50 68 84 80 40 00 E8 7C 00 00 00 6A 08 8D 4C
00001020h: 24 10 68 78 80 40 00 51 E8 33 00 00 00 83 C4 18
00001030h: 33 C0 74 11 68 4C 80 40 00 E8 71 00 00 00 83 C4
00001040h: 04 83 C4 14 C3 68 30 80 40 00 E8 60 00 00 00 83
00001050h: C4 04 83 C4 14 C3 90 90 90 90 90 90 90 90 90 90
```

- Save as serialPatch.exe

SRE Example

```
Command Prompt
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>serialPatch


Enter Serial Number
fjdjfdlfjsd
Serial number is correct.

C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>
```

* **Any** "serial number" now works!
* Very convenient for Trudy!

SRE Example

- Back to IDA Pro disassembly…

serial.exe

```
.text:00401003    push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008    call    sub_4010AF
.text:0040100D    lea     eax, [esp+18h+var_14]
.text:00401011    push    eax
.text:00401012    push    offset aS       ; "%s"
.text:00401017    call    sub_401098
.text:0040101C    push    8
.text:0040101E    lea     ecx, [esp+24h+var_14]
.text:00401022    push    offset aS123n456 ; "S123N456"
.text:00401027    push    ecx
.text:00401028    call    sub_401060
.text:0040102D    add     esp, 18h
.text:00401030    test    eax, eax
.text:00401032    jz      short loc_401045
.text:00401034    push    offset aErrorIncorrect ; "Error! Incorrect serial number.
.text:00401039    call    sub_4010AF
```

serialPatch.exe

```
.text:00401003    push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008    call    sub_4010AF
.text:0040100D    lea     eax, [esp+18h+var_14]
.text:00401011    push    eax
.text:00401012    push    offset aS       ; "%s"
.text:00401017    call    sub_401098
.text:0040101C    push    8
.text:0040101E    lea     ecx, [esp+24h+var_14]
.text:00401022    push    offset aS123n456 ; "S123N456"
.text:00401027    push    ecx
.text:00401028    call    sub_401060
.text:0040102D    add     esp, 18h
.text:00401030    xor     eax, eax
.text:00401032    jz      short loc_401045
.text:00401034    push    offset aErrorIncorrect ; "Error! Incorrect serial number.
.text:00401039    call    sub_4010AF
```

SRE Attack Mitigation

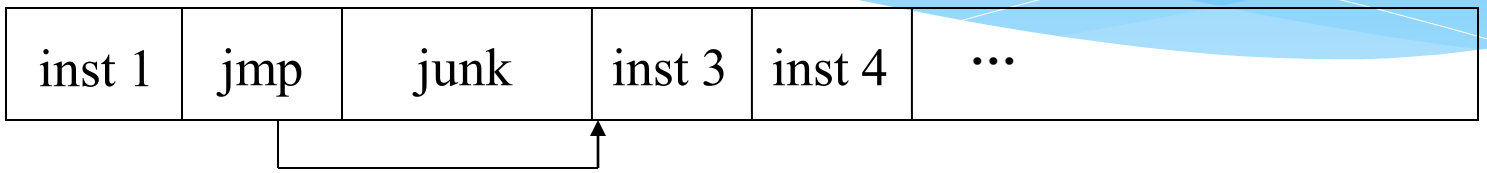
* **Impossible** to prevent SRE on open system
* But can make such attacks more difficult
* **Anti-disassembly techniques**
  * To confuse static view of code
* **Anti-debugging techniques**
  * To confuse dynamic view of code
* **Tamper-resistance**
  * Code checks itself to detect tampering
* Code obfuscation
  * Make code more difficult to understand

Anti-disassembly

* Anti-disassembly methods include
  * Encrypted object code
  * False disassembly
  * Self-modifying code
  * Many others
* Encryption **prevents** disassembly
  * But still need code to decrypt the code!
  * Same problem as with polymorphic viruses

Anti-disassembly Example

* Suppose actual code instructions are

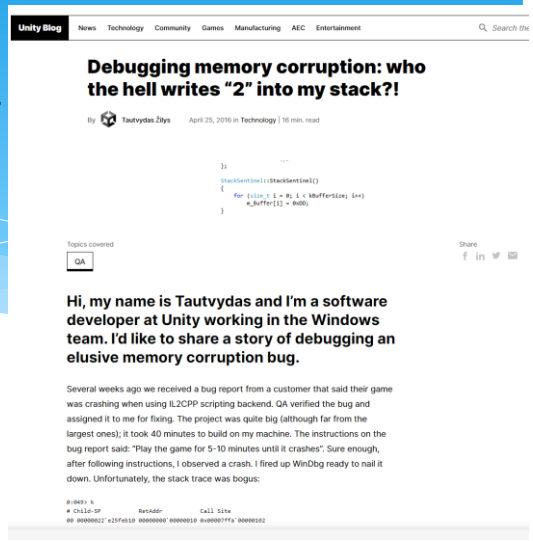


- What the disassembler sees

- This is example of "false disassembly"
- Clever attacker will figure it out!

Anti-debugging

https://blog.unity.com/technology/debugging-memory-corruption-who-the-hell-writes-2-into-my-stack-2


Unity Blog News Technology Community Games Manufacturing AEC Entertainment

Debugging memory corruption: who the hell writes "2" into my stack?!

By Tautvydas Žilys April 25, 2016 in Technology | 16 min. read

```
};

StackSentinel::StackSentinel()
{
    for (size_t i = 0; i < kBufferSize; i++)
        m_Buffer[i] = 0x00;
}
```

Topics covered

QA

Share

Hi, my name is Tautvydas and I'm a software developer at Unity working in the Windows team. I'd like to share a story of debugging an elusive memory corruption bug.

Several weeks ago we received a bug report from a customer that said their game was crashing when using IL2CPP scripting backend. QA verified the bug and assigned it to me for fixing. The project was quite big (although far from the largest ones); it took 40 minutes to build on my machine. The instructions on the bug report said: "Play the game for 5-10 minutes until it crashes". Sure enough, after following instructions, I observed a crash. I fired up WinDbg ready to nail it down. Unfortunately, the stack trace was bogus:

```
0:049> k
# Child-SP          RetAddr           Call Site
00 00000022`e25fe810 00000000`00000010 0x00002ffa`00000102
```

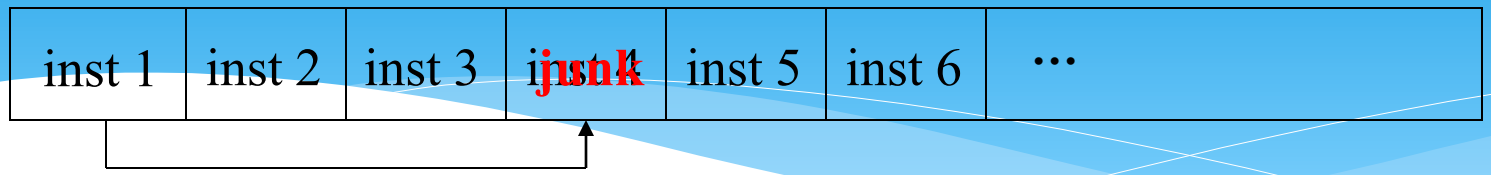
- Monitor for
  - Use of debug registers
  - Inserted breakpoints
- **Debuggers don't handle threads well**
  - Interacting threads may confuse debugger
- Many other debugger-unfriendly tricks
- Undetectable debugger possible in principle
  - Hardware-based debugging (HardICE) is possible

Anti-debugger Example

inst 1	inst 2	inst 3	inst 4	inst 5	inst 6	…

* Suppose when program gets inst 1, it pre-fetches inst 2, inst 3 and inst 4
  * This is done to increase efficiency
* Suppose when debugger executes inst 1, it does **not** pre-fetch instructions
* Can we use this difference to confuse the debugger?

Anti-debugger Example



- Suppose inst 1 **overwrites** inst 4 in memory
- Then program (without debugger) will be OK since it fetched inst 4 at same time as inst 1
- Debugger will be confused when it reaches **junk** where inst 4 is supposed to be
- Problem for program if this segment of code executed more than once!
- Also, code is very platform-dependent
- Again, clever attacker will figure this out!

Tamper-resistance

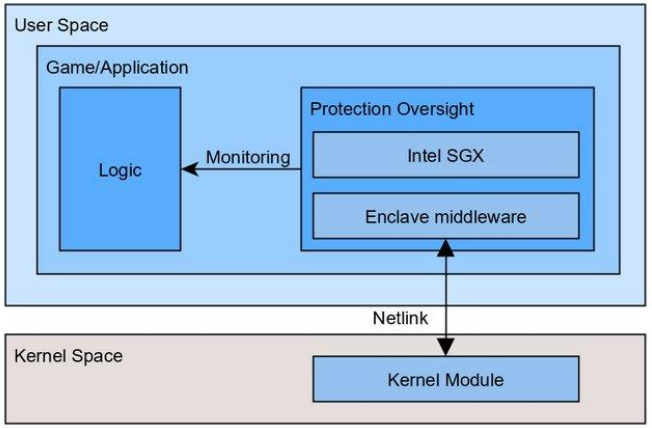
* Goal is to make patching more difficult
* Code can **hash** parts of itself
* If tampering occurs, hash check fails
* Research has shown can get good coverage of code with small performance penalty
* But don't want all checks to look similar
  * Or else easy for attacker to remove checks
* This approach sometimes called "guards"

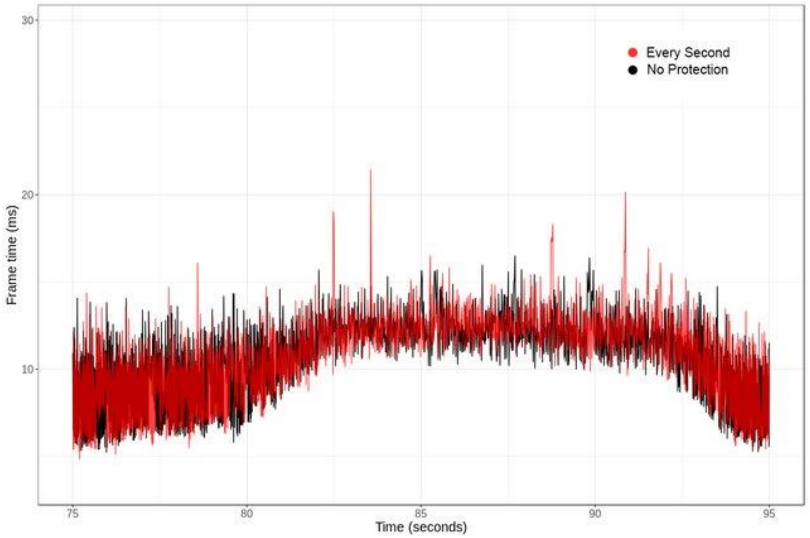
Tamper-resistance

Employment of Secure Enclaves in Cheat Detection Hardening

TrustBus'20

* **SGX-based introspection**





Code Obfuscation

* Goal is to make code hard to understand
* Opposite of good software engineering!
* Simple example: spaghetti code
* Much research into more robust obfuscation
  * Example: **opaque predicate**

```
int x,y
    :
if((x–y)*(x–y) > (x*x–2*x*y+y*y)){…}
```

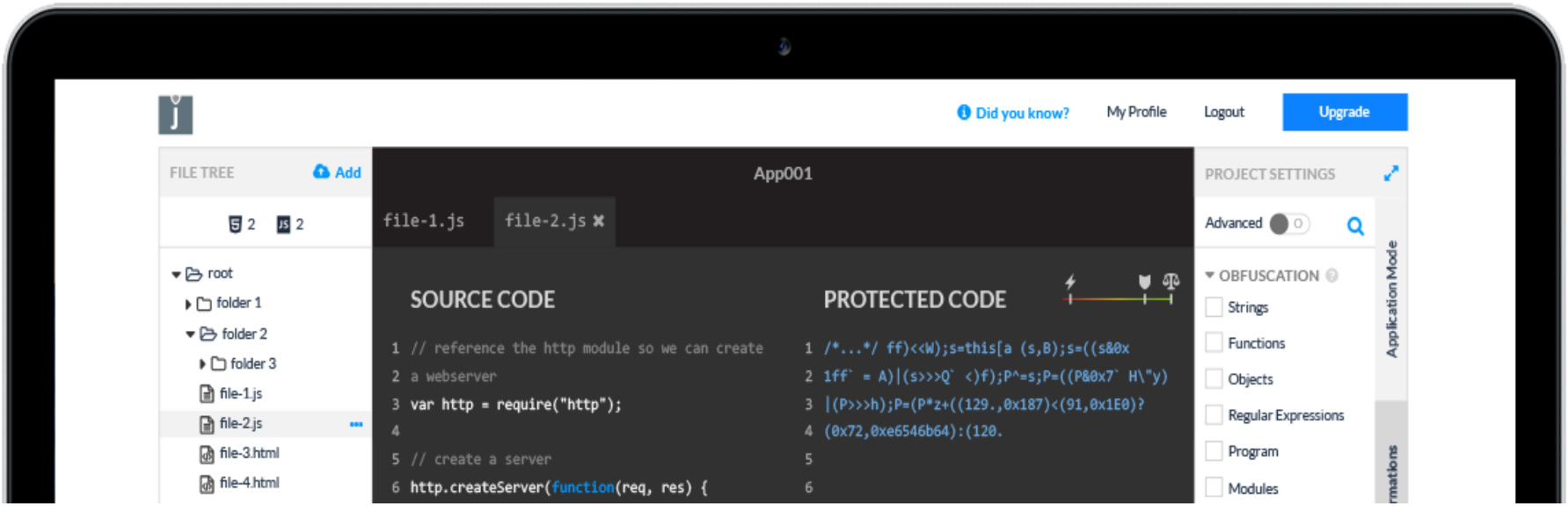
  * The if() conditional is always false
* Attacker will waste time analyzing dead code

Code Obfuscation

* Code obfuscation sometimes promoted as a powerful security technique
* Recently it has been shown that obfuscation probably cannot provide "strong" security
  * On the (im)possibility of obfuscating programs
  * Some question significance of result (Thomborson)
* Obfuscation might still have practical uses!
  * Even if it can never be as strong as crypto

Example - JSCRAMBLER

* Jscrambler can make your javascript application become self-defensive. If Jscrambler detects that your code was tampered or if suspicious debugging activities are in action it can make the code derail the execution of the program in a standalone way. (https://jscrambler.com)



Authentication Example

* Software used to determine authentication
* Ultimately, authentication is 1-bit decision
  * Regardless of method used (pwd, biometric, …)
* Somewhere in authentication software, a single bit determines success/failure
* If attacker can find this bit, he can force authentication to always succeed
* Obfuscation makes it more difficult for attacker to find this all-important bit

Obfuscation

* Obfuscation forces attacker to analyze larger amounts of code
* Method could be combined with
  * Anti-disassembly techniques
  * Anti-debugging techniques
  * Code tamper-checking
* All of these increase work (and pain) for attacker
* But a persistent attacker will ultimately win

Software Cloning (BOBE)

* Suppose we write a piece of software
* We then distribute an identical copy (or clone) to each customers
* If an attack is found on one copy, the same attack works on all copies
* This approach has no resistance to “break once, break everywhere” (BOBE)
* This is the usual situation in software development

Metamorphic Software

* Metamorphism is used in malware
* Can metamorphism also be used for good?
* Suppose we write a piece of software
* Each copy we distribute is different
  * This is an example of metamorphic software
* Two levels of metamorphism are possible
  * All instances are functionally distinct (only possible in certain application)
  * All instances are functionally identical but differ internally (always possible)
* We consider the latter case

Metamorphic Software

* If we distribute N copies of cloned software
  * One successful attack breaks all N
* If we distribute N metamorphic copies, where each of N instances is functionally identical, but they differ internally…
  * An attack on one instance does not necessarily work against other instances
  * In the best case, N times as much work is required to break all N instances

Metamorphic Software

* **We cannot prevent SRE attacks**
* The best we can hope for is BOBE ("break once, break everywhere") resistance
* Metamorphism can improve BOBE resistance
* Consider the analogy to genetic diversity
  * If all plants in a field are genetically identical, one disease can kill **all** of the plants
  * If the plants in a field are genetically diverse, one disease can only kill **some** of the plants

Cloning vs Metamorphism

* Suppose our software has a buffer overflow
* **Cloned** software
  * Same buffer overflow attack will work against **all** cloned copies of the software
* **Metamorphic** software
  * Unique instances — all are functionally the same, but they differ in internal structure
  * Buffer overflow likely exists in all instances
  * But a specific buffer overflow attack will only work against **some** instances
  * Buffer overflow attacks are delicate!