xorlab

# Compilers, Memory Errors and Hardening Techniques

Advanced Compiler Design, 18.5.2016

Antonio Hüseyin Barresi
Co-founder & CEO
xorlab AG

Spinoff ETH zürich

LST Laboratory for Software Technology

# Executive Opinion Survey 2015


http://knowledge.zurich.com/risk-interconnectivity/time-to-wake-up-to-the-risks-of-cyber-crime

# What does that have to do with…

# Compilers?

# This talk

- Cyber attacks and compilers
- Memory errors and exploitation
- Hardening techniques and run-time security

# About

**Antonio Barresi**

antonio.barresi@xorlab.com

@AntonioHBarresi

Co-founder of

xorlab

https://www.xorlab.com

Spinoff ETH zürich

LST Laboratory for Software Technology

CSIS | CENTER FOR STRATEGIC & INTERNATIONAL STUDIES

# Cyber attacks cost the world economy >**400** $ billion yearly

Net losses: Estimating the Global Cost of Cybercrime, Economic impact of cybercrime II, Center for Strategic and International Studies, June 2014

CSIS | CENTER FOR STRATEGIC & INTERNATIONAL STUDIES

# Cyber crime is a growth industry

Net losses: Estimating the Global Cost of Cybercrime, Economic impact of cybercrime II, Center for Strategic and International Studies, June 2014

CSIS | CENTER FOR STRATEGIC & INTERNATIONAL STUDIES

- "Cybercrime is a tax on innovation and slows the pace of global innovation"
- Direct financial losses, loss of competitiveness and other impact
- High-income countries suffer the most
  - 0.9% of GDP in average
  - Loss of 5.8 Bn CHF (CH 2013, GDP 646 Bn CHF)

Net losses: Estimating the Global Cost of Cybercrime, Economic impact of cybercrime II, Center for Strategic and International Studies, June 2014

# Example

# 100 banks, 30 countries, 2 years

# $1 Billion

BUSINESS HACKING

## Hackers Steal $1 Billion in Massive, Worldwide Breach

Matt Vella @mattvella | Feb. 15, 2015

**A prominent cybersecurity firm says that thieves have infiltrated more than 100 banks in 30 countries over the past two years**

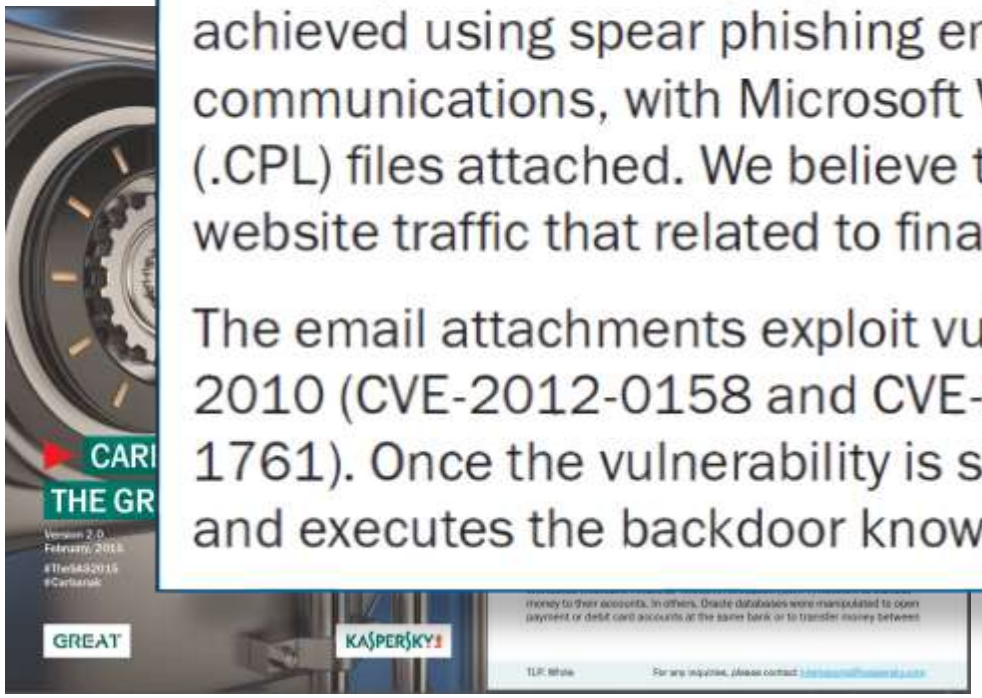
Hackers have stolen as much as $1 billion from banks around the world, according to a prominent cybersecurity firm. In a report scheduled to be delivered Monday, Russian security company Kaspersky Lab claims that a hacking ring has infiltrated more than 100 banks in 30 countries over the past two years.

Kaspersky says digital thieves gained access to banks' computer systems through phishing schemes and other confidence scams. Hackers then lurked in the institutions' systems, taking screen shots or even video of employees at work. Once familiar with the banks' operations, the hackers could steal funds without raising alarms, programming ATMs to dispense money at specific times for instance or transferring funds to fraudulent accounts. First outlined by the

# The great bank robbery


https://securelist.com/blog/research/68732/the-great-bank-robbery-the-carbanak-apt/

# The great bank robbery

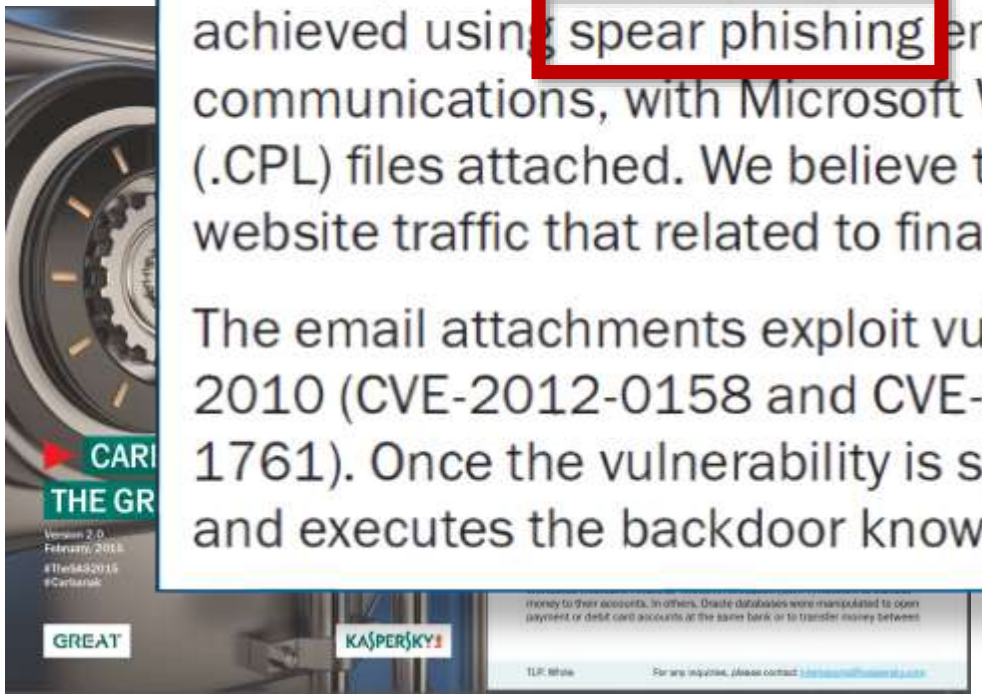



An analysis of the campaign has revealed that the initial infections were achieved using spear phishing emails that appeared to be legitimate banking communications, with Microsoft Word 97 – 2003 (.doc) and Control Panel Applet (.CPL) files attached. We believe that the attackers also redirected to exploit kits website traffic that related to financial activity.

The email attachments exploit vulnerabilities in Microsoft Office 2003, 2007 and 2010 (CVE-2012-0158 and CVE-2013-3906) and Microsoft Word (CVE-2014-1761). Once the vulnerability is successfully exploited, the shellcode decrypts and executes the backdoor known as **Carbanak**.

https://securelist.com/blog/research/68732/the-great-bank-robbery-the-carbanak-apt/

# The great bank robbery

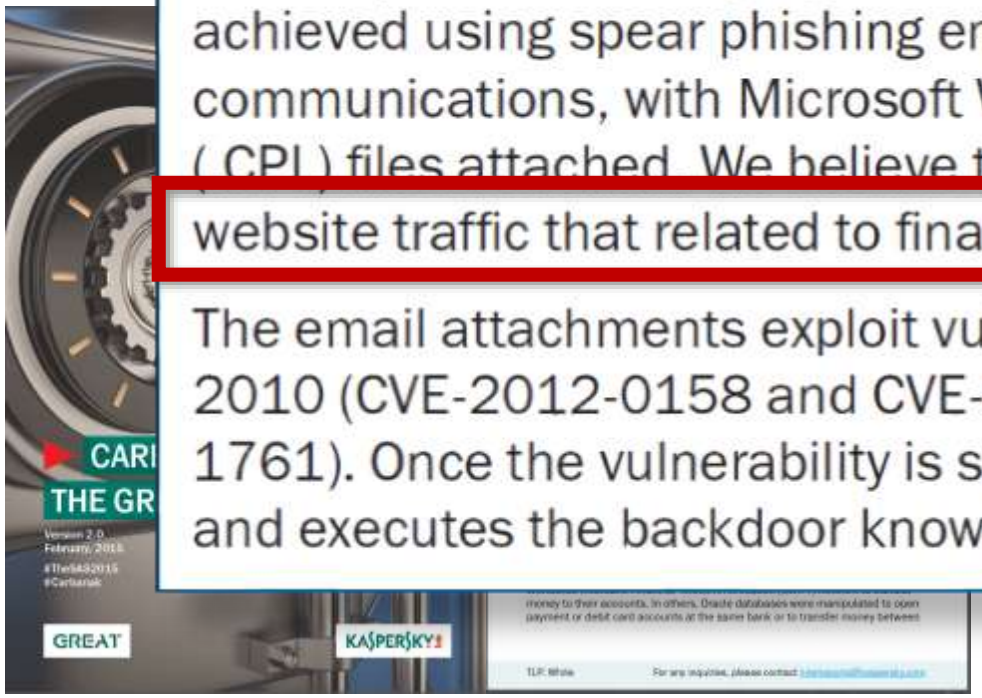



An analysis of the campaign has revealed that the initial infections were achieved using spear phishing emails that appeared to be legitimate banking communications, with Microsoft Word 97 – 2003 (.doc) and Control Panel Applet (.CPL) files attached. We believe that the attackers also redirected to exploit kits website traffic that related to financial activity.

The email attachments exploit vulnerabilities in Microsoft Office 2003, 2007 and 2010 (CVE-2012-0158 and CVE-2013-3906) and Microsoft Word (CVE-2014-1761). Once the vulnerability is successfully exploited, the shellcode decrypts and executes the backdoor known as **Carbanak**.

https://securelist.com/blog/research/68732/the-great-bank-robbery-the-carbanak-apt/

# The great bank robbery

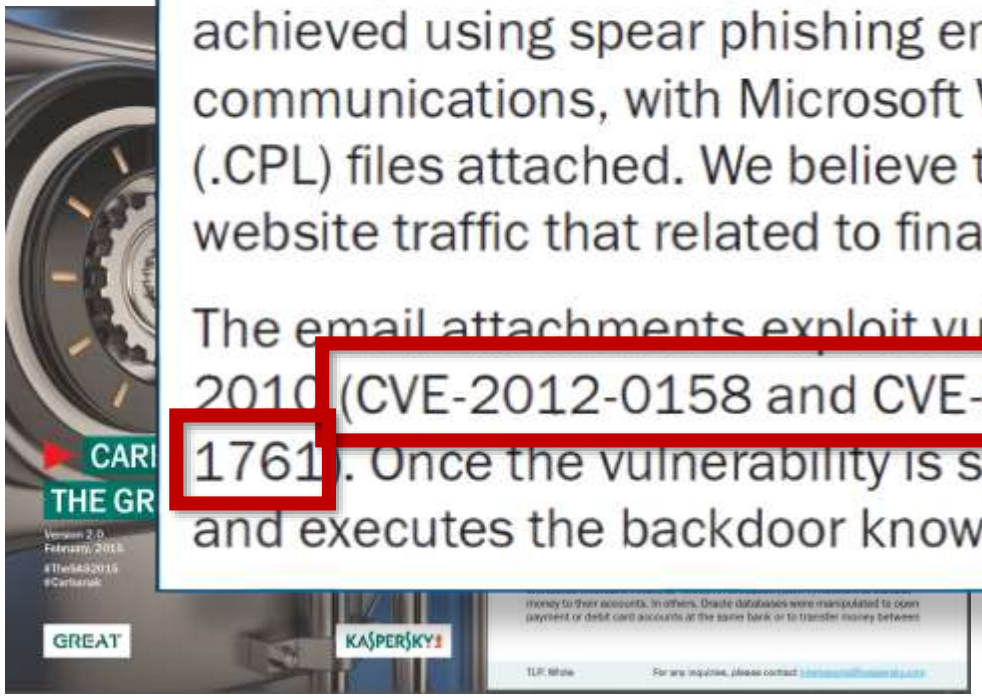



An analysis of the campaign has revealed that the initial infections were achieved using spear phishing emails that appeared to be legitimate banking communications, with Microsoft Word 97 – 2003 (.doc) and Control Panel Applet (.CPL) files attached. We believe that the attackers also redirected to exploit kits website traffic that related to financial activity.

The email attachments exploit vulnerabilities in Microsoft Office 2003, 2007 and 2010 (CVE-2012-0158 and CVE-2013-3906) and Microsoft Word (CVE-2014-1761). Once the vulnerability is successfully exploited, the shellcode decrypts and executes the backdoor known as **Carbanak**.

https://securelist.com/blog/research/68732/the-great-bank-robbery-the-carbanak-apt/

# The great bank robbery

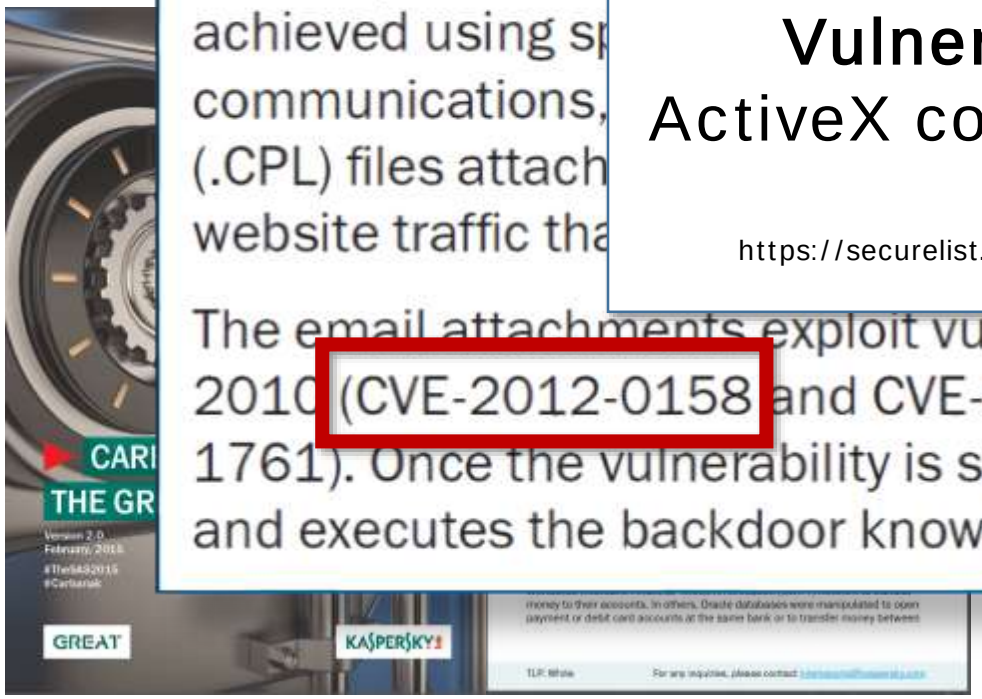



An analysis of the campaign has revealed that the initial infections were achieved using spear phishing emails that appeared to be legitimate banking communications, with Microsoft Word 97 – 2003 (.doc) and Control Panel Applet (.CPL) files attached. We believe that the attackers also redirected to exploit kits website traffic that related to financial activity.

The email attachments exploit vulnerabilities in Microsoft Office 2003, 2007 and 2010 (CVE-2012-0158 and CVE-2013-3906) and Microsoft Word (CVE-2014-1761). Once the vulnerability is successfully exploited, the shellcode decrypts and executes the backdoor known as **Carbanak**.

https://securelist.com/blog/research/68732/the-great-bank-robbery-the-carbanak-apt/

# The great bank robbery




An analysis of the
achieved using sp
communications,
(.CPL) files attach
website traffic tha

The email attachments exploit vulnerabilities in Microsoft Office 2003, 2007 and 2010 (CVE-2012-0158 and CVE-2013-3906) and Microsoft Word (CVE-2014-1761). Once the vulnerability is successfully exploited, the shellcode decrypts and executes the backdoor known as **Carbanak**.

„CVE-2012-0158 is a **buffer overflow Vulnerability** in the ListView / TreeView ActiveX controls in the MSCOMCTL.OCX library.“

https://securelist.com/analysis/publications/37158/the-curious-case-of-a-cve-2012-0158-exploit/

https://securelist.com/blog/research/68732/the-great-bank-robbery-the-carbanak-apt/

# Exploit and Zero-Day Attack

Exploit = “An **exploit** is a **piece of software, a chunk of data**, or a sequence of commands that **takes advantage of a bug**, glitch or vulnerability in order to...“

Zero-Day Attack = “A **zero-day** attack or threat is an attack that exploits a **previously unknown vulnerability** in a computer application, ...“

http://en.wikipedia.org/wiki/Exploit_(computer_security)
http://en.wikipedia.org/wiki/Zero-day_attack

# Software attacks

- Attacking software is an important capability for attackers
  - Scales well
  - Can be done remotely
  - Very powerful

# Arbitrary code-execution

# Goal of an attacker


THE UNITED STATES OF AMERICA
DF 20493661 B
100
ONE HUNDRED DOLLARS
FEDERAL RESERVE NOTE
THIS NOTE IS LEGAL TENDER FOR ALL DEBTS, PUBLIC AND PRIVATE

# Most important thing

Profitability

# Profit

Revenue –
(Costs + expenses + taxes)

# Cyber criminals

- Have costs and expenses too!
- An economic decision
- Fighting cyber crime means limiting its profitability

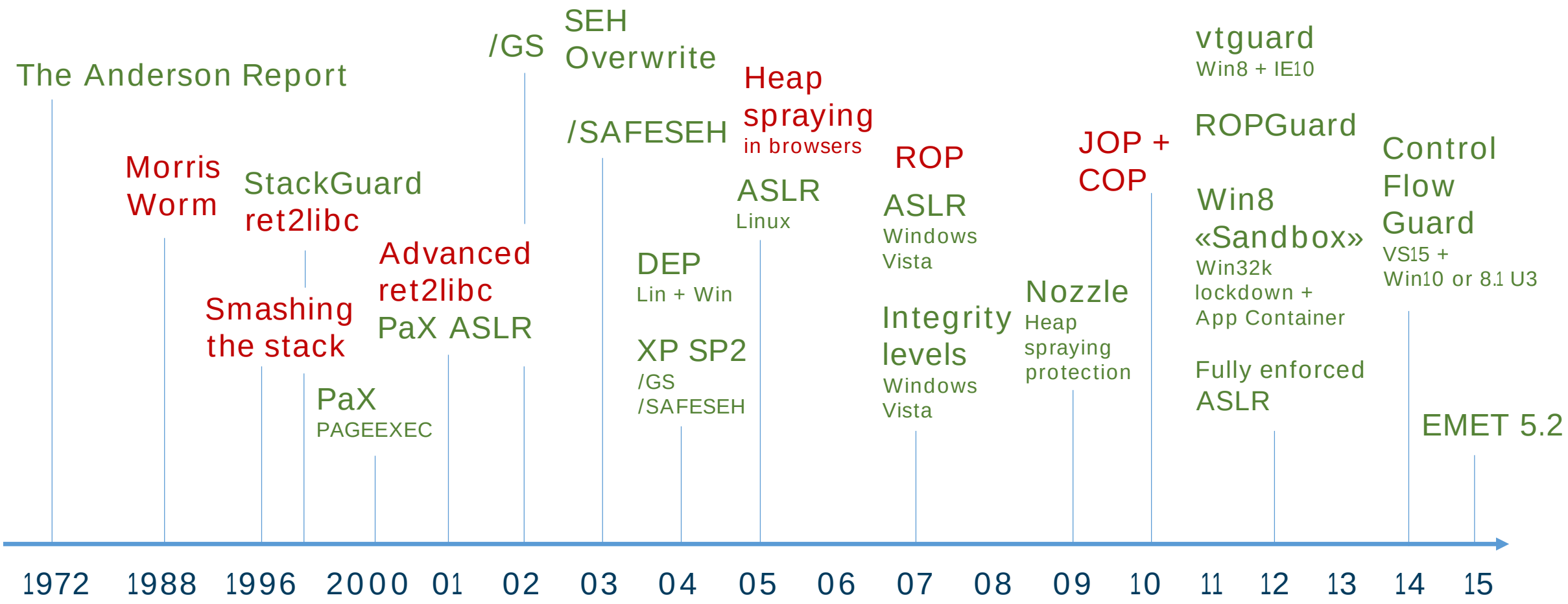
# Cyber criminals

- Have costs and expenses too!
- An economic decision
- Fighting cyber crime means limiting it's profitability

# Software hardening

- Compile and run-time techniques
- To make software exploitation harder
- But also slower

# Exploit mitigations since the 90s



# Attacker Return

$$\text{Attacker Return} = \left(\begin{array}{c}\text{Gains per use}\\ \times \\ \text{Opportunities to use}\end{array}\right) - \left(\begin{array}{c}\text{Cost to acquire vulnerability}\\ + \\ \text{Cost to weaponize}\end{array}\right)$$

Mitigating Software Vulnerabilities, July 2011, Microsoft

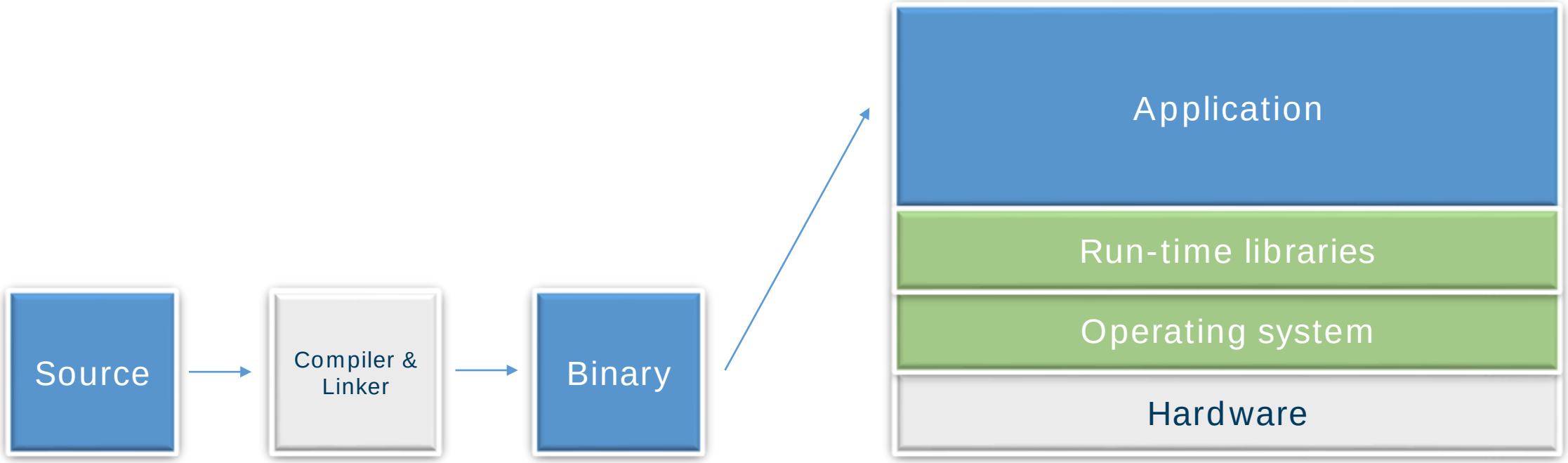
# Attacker Return

$$\text{Attacker Return} = \left(\text{Gains per use} \times \text{Opportunities to use}\right) - \left(\text{Cost to acquire vulnerability} + \text{Cost to weaponize}\right)$$

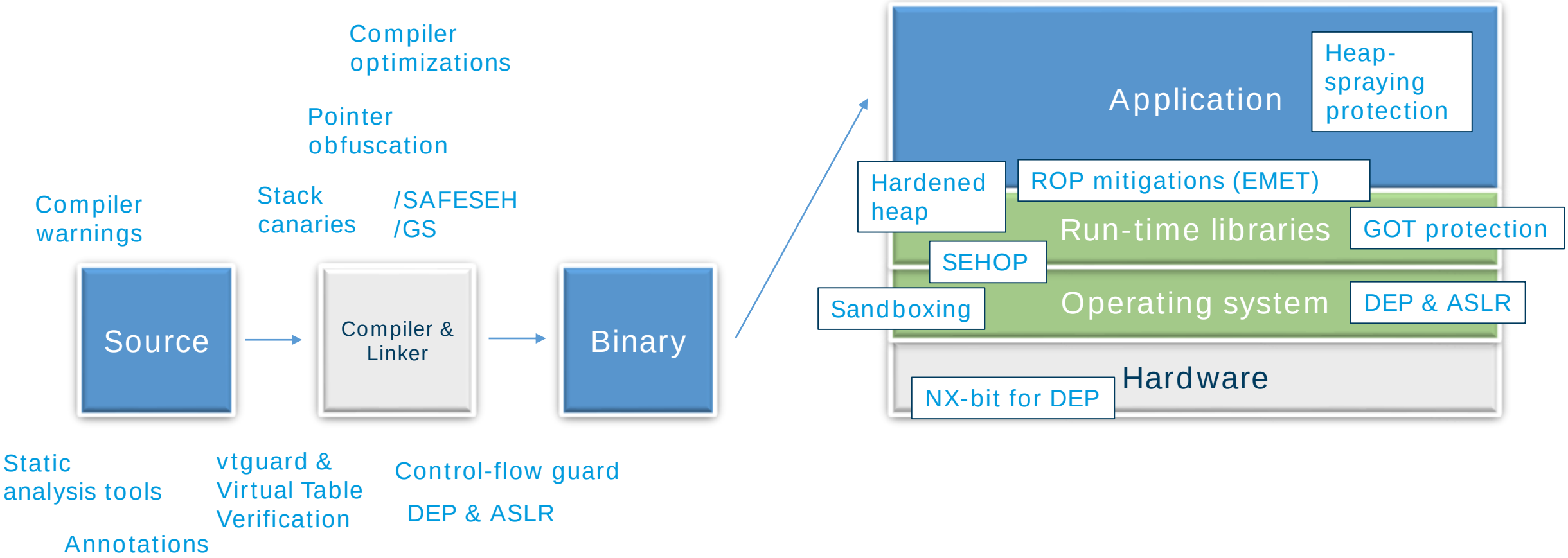
Increase costs

Mitigating Software Vulnerabilities, July 2011, Microsoft

# Hardening value chain

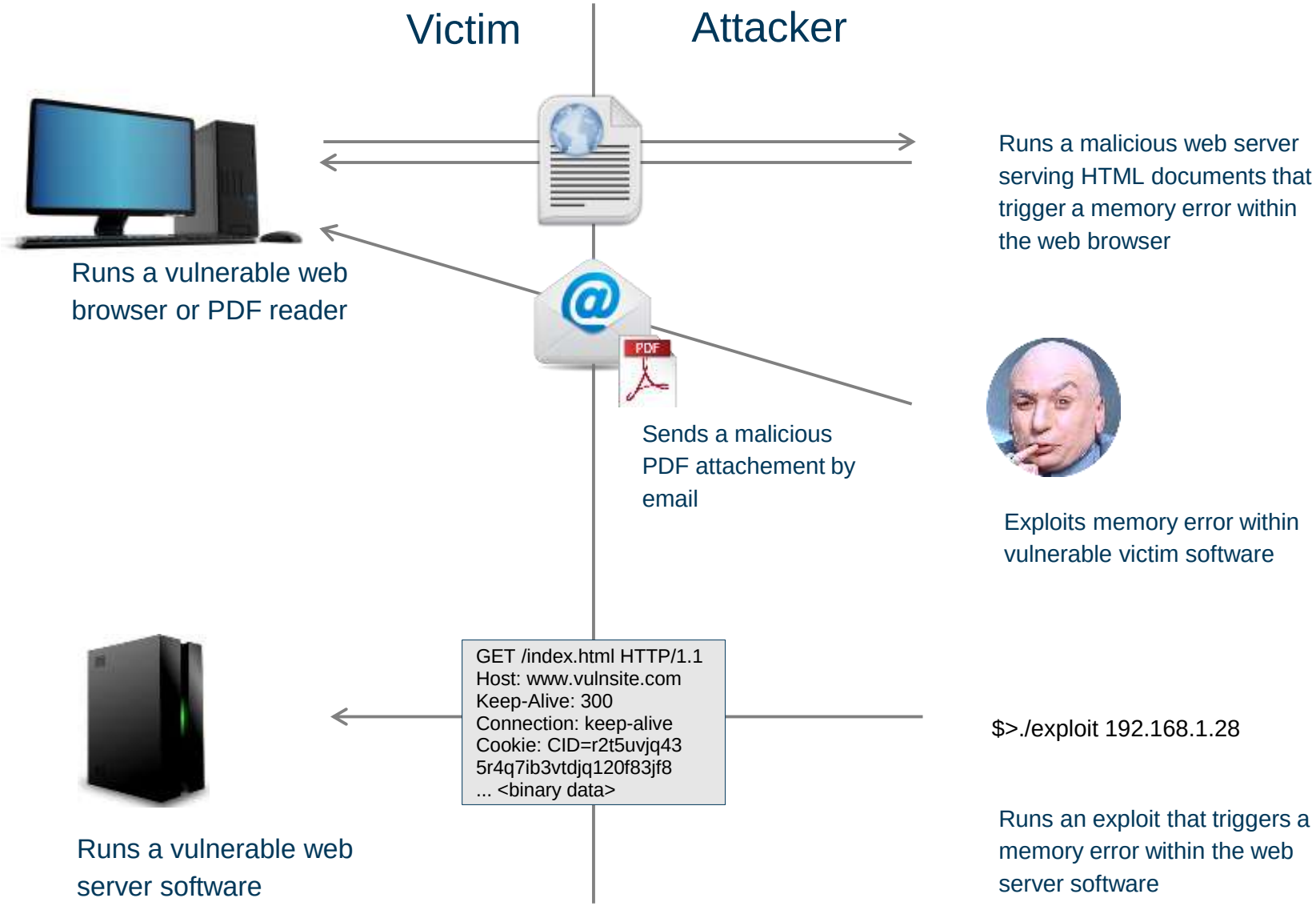


# Hardening value chain



# Software attacks

# Attacks


Victim
Attacker
Runs a malicious web server serving HTML documents that trigger a memory error within the web browser
Runs a vulnerable web browser or PDF reader
PDF
Sends a malicious PDF attachement by email
Exploits memory error within vulnerable victim software
GET /index.html HTTP/1.1
Host: www.vulnsite.com
Keep-Alive: 300
Connection: keep-alive
Cookie: CID=r2t5uvjq43
5r4q7ib3vtdjq120f83jf8
... <binary data>
$>./exploit 192.168.1.28
Runs a vulnerable web server software
Runs an exploit that triggers a memory error within the web server software

# Memory corruption

- Exist in programs written in “unsafe“ languages
  - C/C++

- Two basic types exist
  - Spatial and temporal memory errors

- Allow attackers to corrupt memory in a more or less controllable way

# «Unsafe» languages

Allow low-level access to memory

- Typed pointers & pointer arithmetic
- No automatic bounds checking / index checking

Weakly enforced typing

- Cast (almost) anything to pointers

Explicit memory management

- Like malloc() & free() in C

# «Unsafe» languages - C

```c
#include <stdio.h>
#include <math.h>

long computation(int x, int y, int z, double f) {
 return (long)(((x*y)/z)*sin(f));
}

void main()
{
  *(int*)computation(32, 64, 2, M_PI/6) = 128;
  return;
}
```

```
shell:~$ gcc -o segfault segfault.c -lm
shell:~$ ./segfault
Segmentation fault (core dumped)
shell:~$
```

# «Unsafe» languages - C

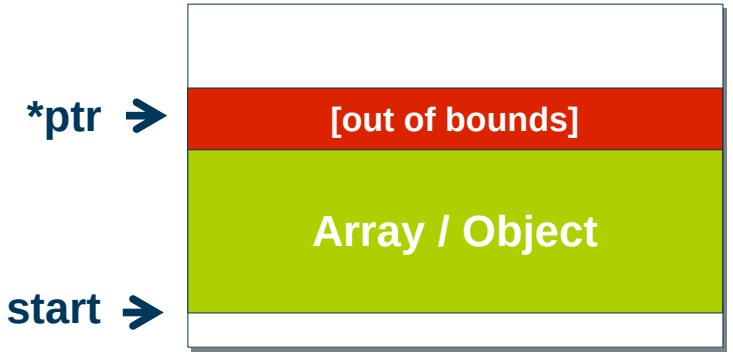
```c
#include <stdio.h>
#include <string.h>

#define STDIN 0

void vulnFunc() {
        char buf[1024];
        read(STDIN, buf, 2048);
}
```

```c
void main() {
        printf("read> ");
        vulnFunc();
        return;
}
```

```
shell:~$ gcc -o vuln vuln.c -fno-stack-protector
shell:~$ ./vuln
read> hi there!
shell:~$ ./vuln
read> AAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAA...
Segmentation fault (core dumped)
shell:~$
```

# Types of memory errors

## Spatial error



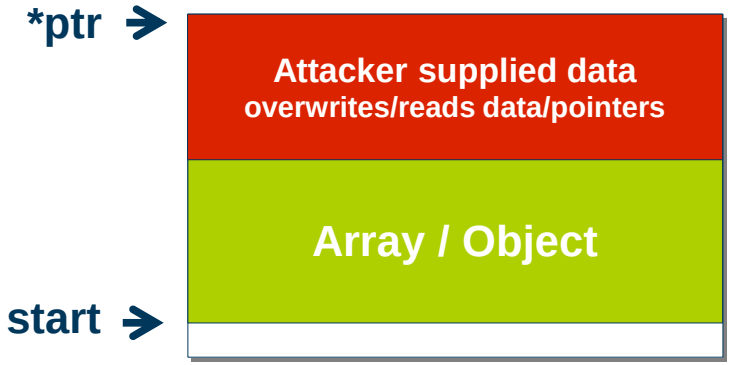

- De-reference pointer that is out of bounds

## Temporal error


- De-reference pointer to freed memory

# Exploiting memory errors

## Spatial error




- Overwrite data or pointers
- Used or de-referenced later

## Temporal error


- Make application allocate memory in the freed area
- Used as old type

# Attackers use memory errors to

Overwrite data or pointers

- That might be used to overwrite data or pointers
- Function pointers, sensitive data, index values, control-flow sensitive data etc.

Leak information

- E.g., corrupt a length field

**Construct attacker primitives**

- Write primitive (write any value to arbitrary address)
- Read primitive (read from any address)

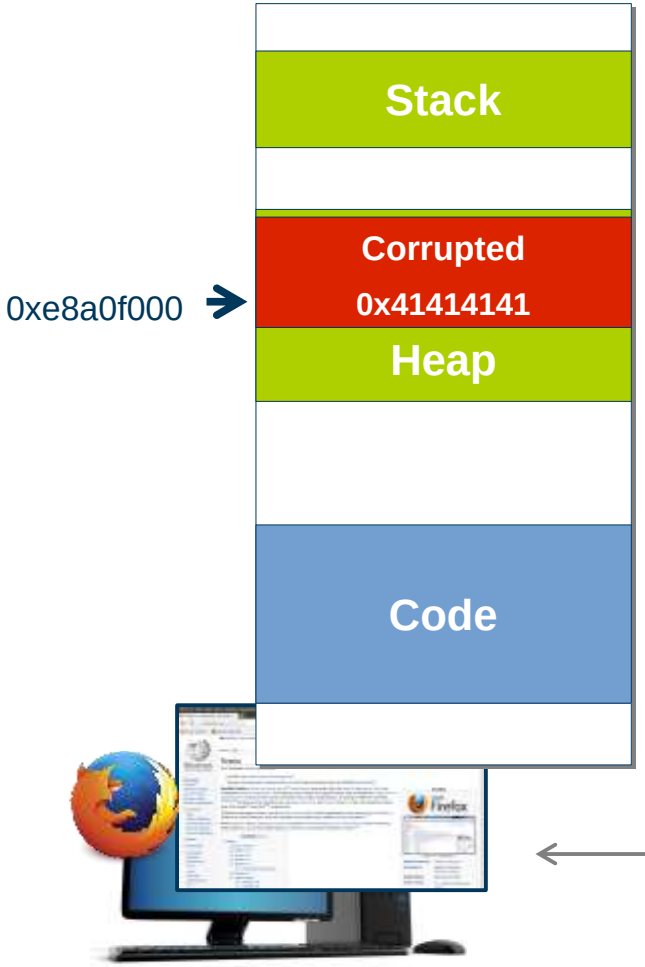
# Types of bugs

- Out-of-bounds bugs / Buffer overflows
  - On stack or heap
- Dangling pointer / Use-after-free
- Integer bugs, signedness bugs
- Format string bugs
- Uninitialized memory
- NULL pointer dereference

# Memory corruption attack

Victim

Attacker




```
<html>
<head>
</head>
<body>
... <html or javascript that triggers the memory corruption vulnerability> ...
</body>
</html>
```

## Write primitive

*0xe8a0f000 = 0x41414141

# Attack types

- Code corruption attack
- **Control-flow hijack attack**
- Data-only attack
- Information leak

Attack model according to: "SoK: Eternal War in Memory" Laszlo Szekeres, Mathias Payer, Tao Wei, Dawn Song
http://www.cs.berkeley.edu/~dawnsong/papers/Oakland13-SoK-CR.pdf

# Control-flow hijack attack

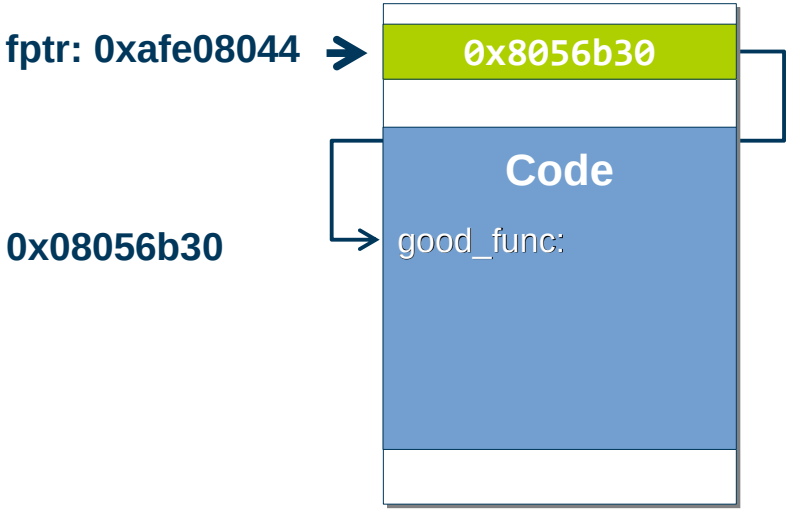
- Most powerful attack
- Hijack control-flow
  - To attacker supplied arbitrary machine code
  - To existing code (code-reuse attack)
- Corrupt code pointers
  - Return addresses, function pointers, vtable entries, exception handlers, jmp_bufs

# Control-flow hijack attack

Most ISAs support indirect branch instructions

E.g., x86 ret, indirect jmp, indirect call

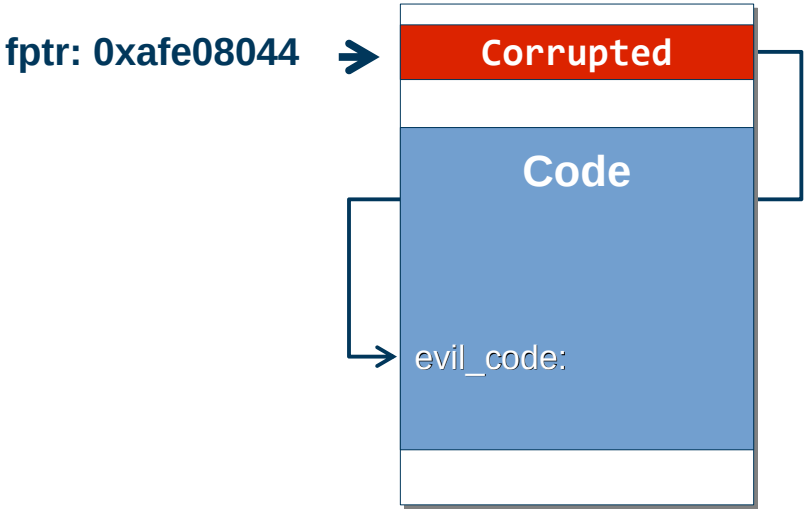
branch *fptr



# Control-flow hijack attack

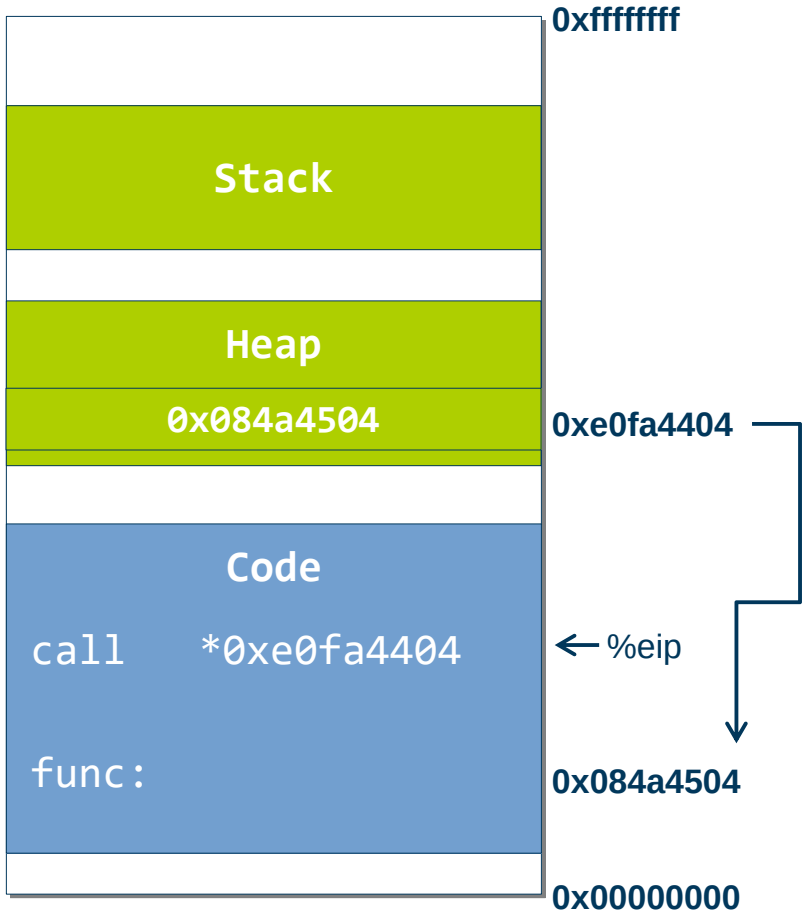
fptr corrupted by attacker

branch *fptr

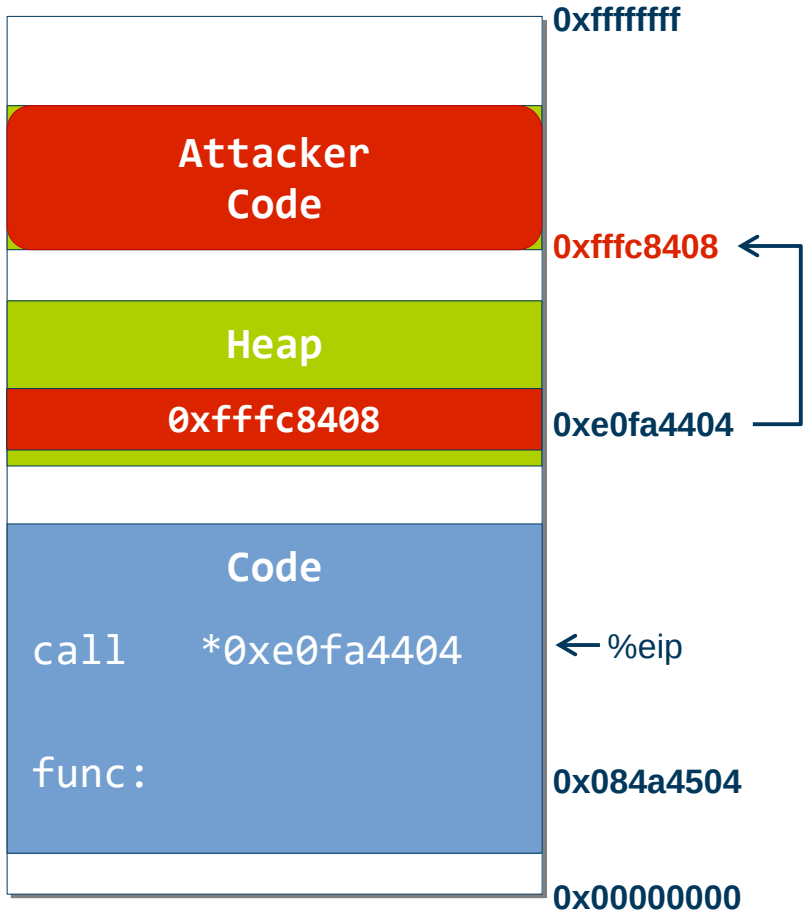



**Attacker goal**: hijack control-flow to injected machine code or to “evil functions“

# Control-flow hijack attack




Indirect call to func()



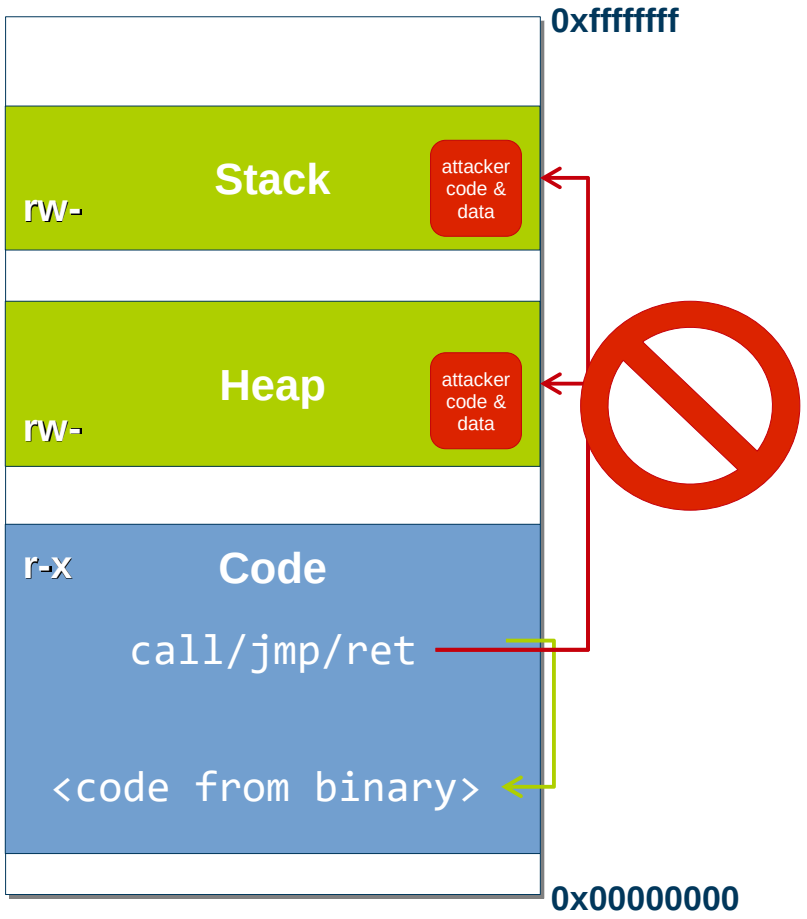

Hijacked indirect call

# Non-eXecutable data (NX)

Make data regions non-executable (by default)

Changing protection flags or allocating rwx memory still possible

Required for JITs



# Non-eXecutable data (NX)

Code regions will be non-writable

- Else code corruption attacks possible

Also known as

- Data Execution Prevention (DEP) on Windows
- NX, Non-eXecutable Memory on Linux
- W^X, on OpenBSD

Implemented by hardware where available (NX-bit)

# NX / DEP – implementation issues

Binary images need to provide separate sections / segments that can be mapped exclusively as

- rw- OR r-x
- Linker support required

Linker support

Self-modifying code not allowed

- Compiler support required
- If code is generated just-in-time, explicit rwx allocation required

Compiler support

# Bypassing NX / DEP

Only use existing code

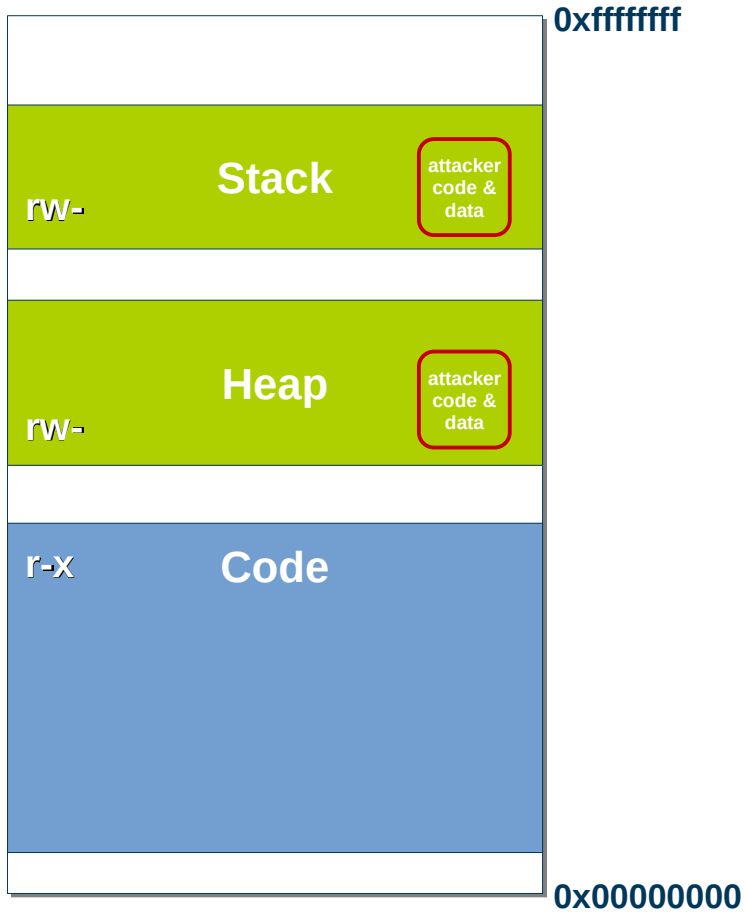
Code-reuse attack

- ret2libc, ret2bin, ret2*
- Return-oriented programming (ROP)
- Jump/Call-oriented programming

Use code-reuse technique to change protection flags

Make memory executable

- mprotect/VirtualProtect
- mmap/VirtualAlloc



# Code-reuse attack

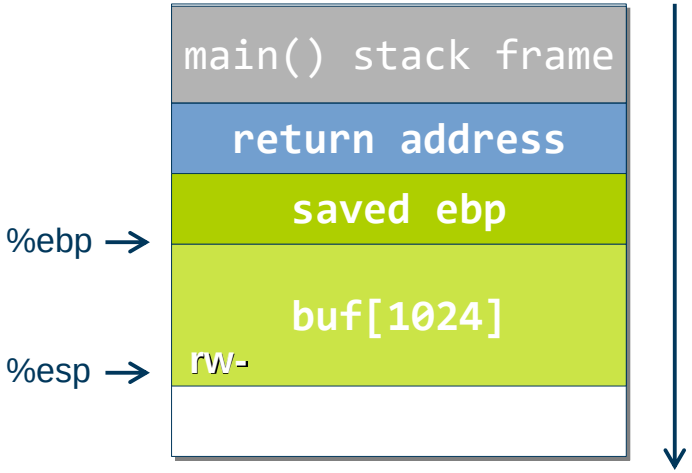
```
void vulnFunc() {
        char buf[1024];
        read(STDIN, buf, 2048);
}
```

Stack-based buffer overflow

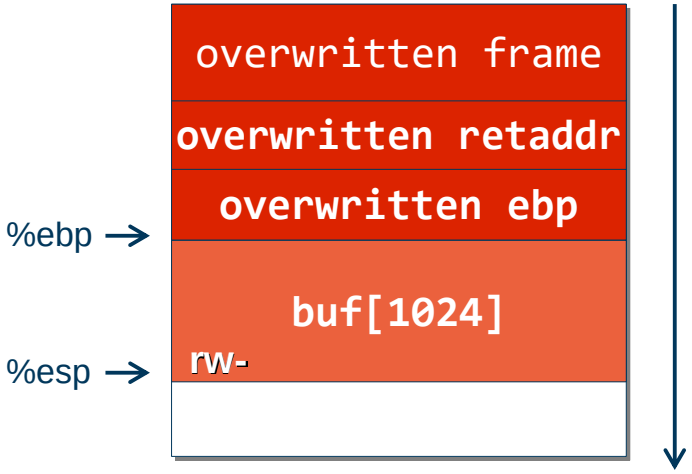
%eip and %ebp under control

Local variables and buffers under attacker control

Stack during vulnFunc()




Stack after read()



# Code-reuse attack

Before NX/DEP

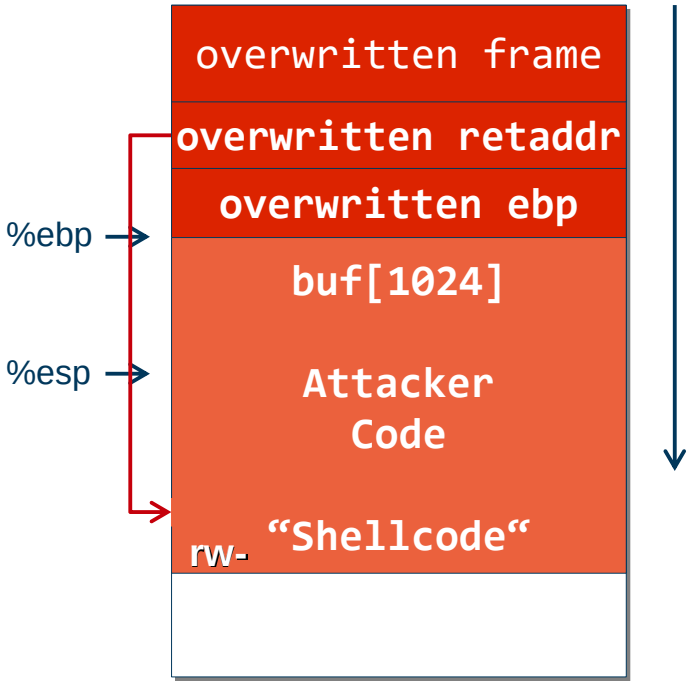
Return to attacker supplied code

⟹ Shellcode

Use existing code

Executable or libraries
E.g., mprotect()

Stack after read()



# Code-reuse attack

## Let's call mprotect()

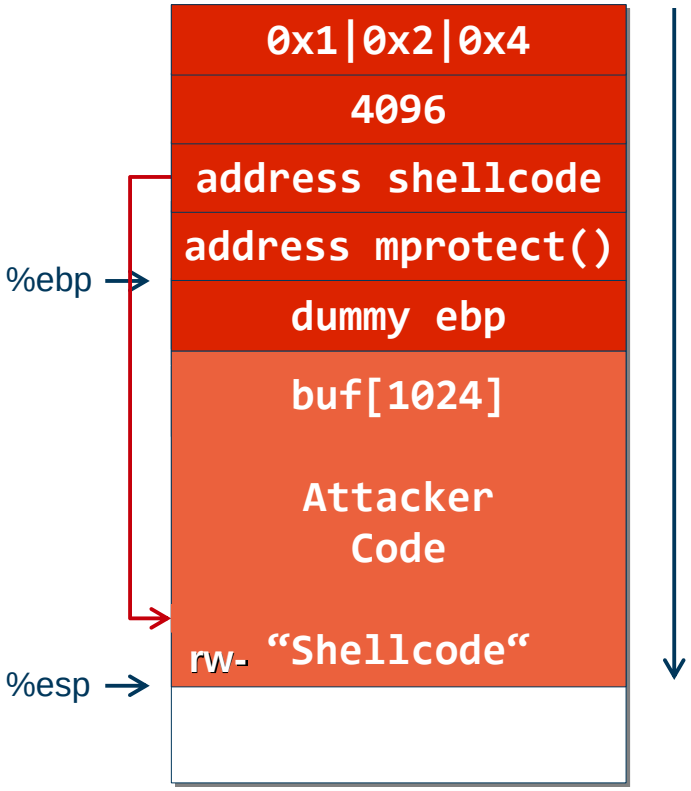
mprotect(address_shellcode, 4096, 0x1|0x2|0x4)

0x1 | 0x2 | 0x4 = RWX

## mprotect() will make the stack executable

...and return to our shellcode

Stack after read()



# Code-reuse attack

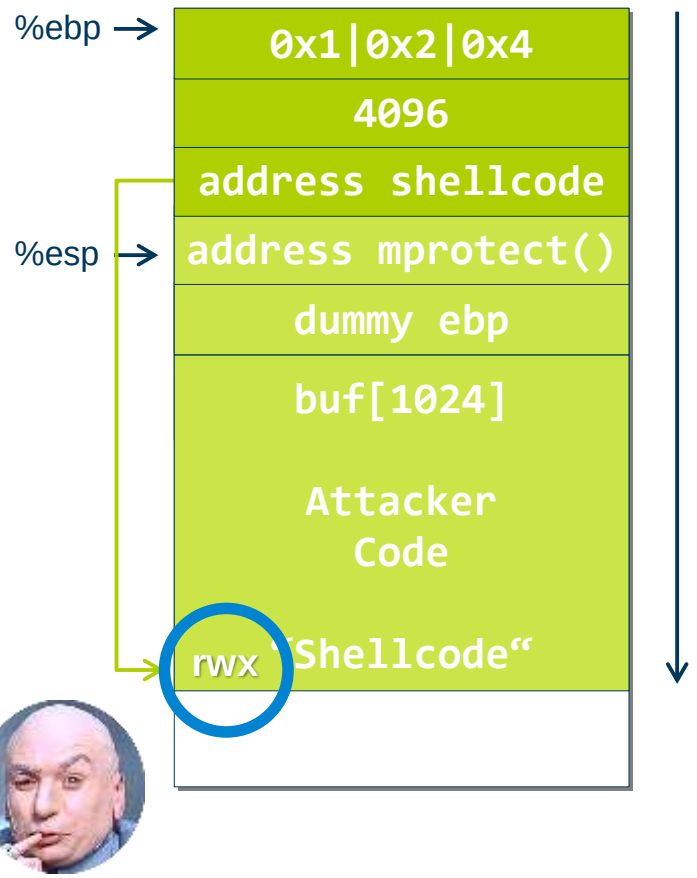
After mprotect() our stack is executable again

mprotect() will return to our shellcode

x64 or ARM:

Fill registers with parameters before invoking mprotect()

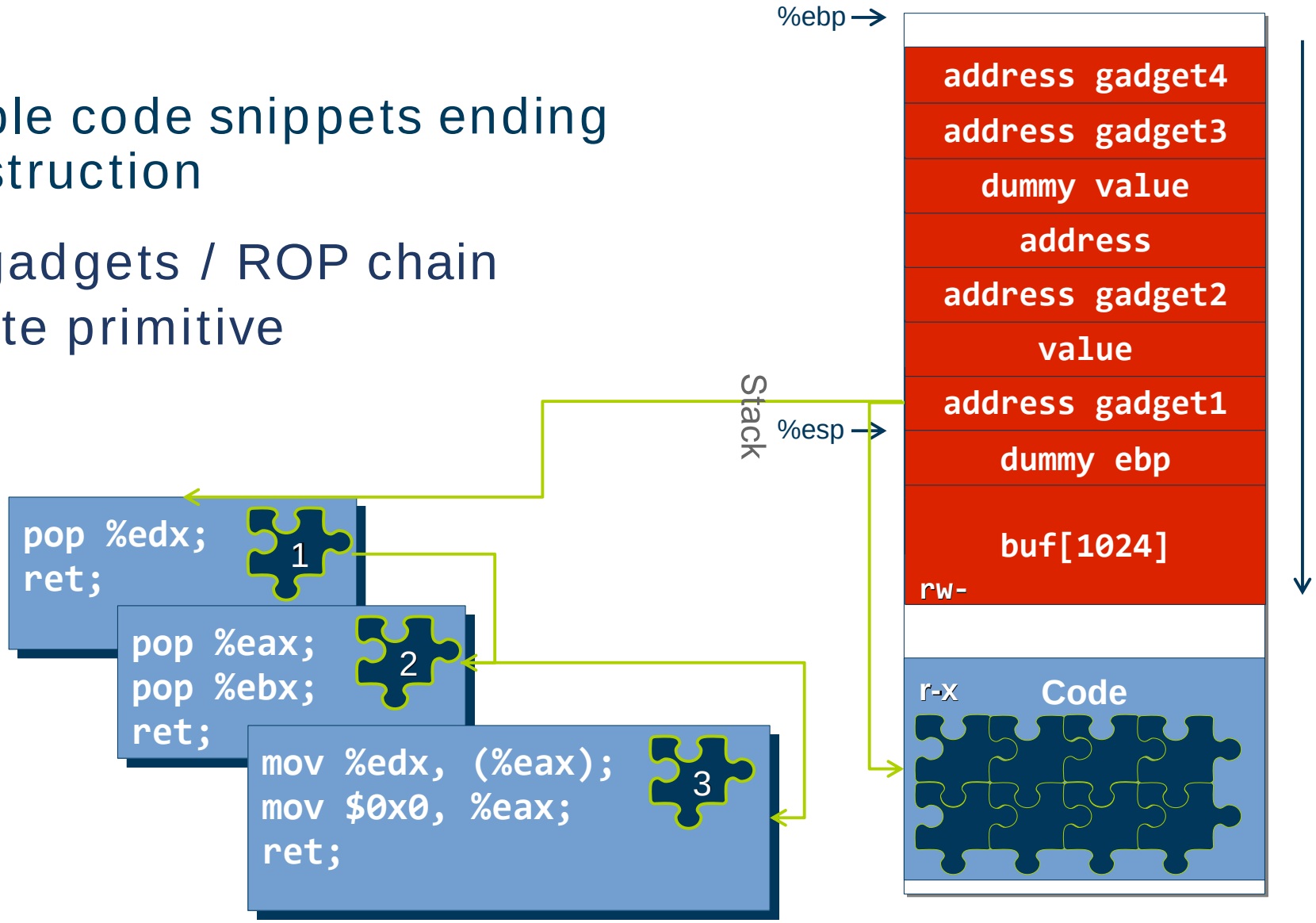
Stack after mprotect()



# Return-oriented programming

Use available code snippets ending with ret instruction

- Called gadgets / ROP chain
- E.g., write primitive



# Return-oriented programming

- Turing complete although not required
- Can be initiated over call or jmp as well
- Need to be in control of memory %esp is pointing to

  Or make %esp point to area under control
- Also possible with jmp or call gadgets

# Addresses in memory

To hijack control-flow or to corrupt memory an attacker needs to know where things are in memory

- Addresses of data or pointers to corrupt
- Addresses of injected shellcode/payload
- Addresses of gadgets

# Addresses in memory

Once upon a time…

Addresses were more or less predictable

Executables and libraries were prelinked to certain addresses

Stack and Heap base addresses were fixed

- With differences at runtime for specific locations due to the dynamic behaviour of the process

# ASLR

Today most Operating Systems implement
**Address Space Layout Randomization**

Randomize memory layout of processes to make address prediction or guessing hard

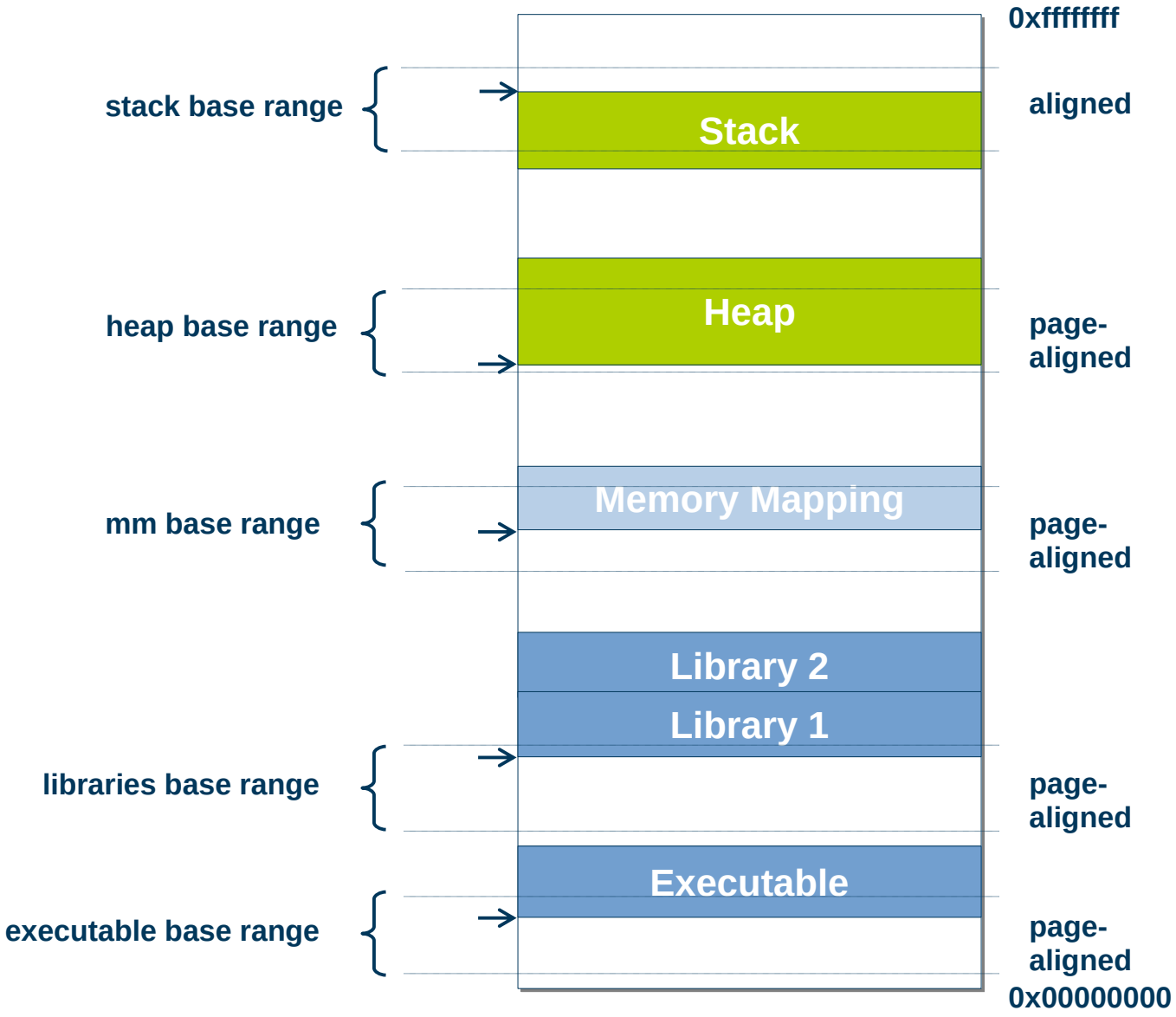
What can be randomized?

- OS: Stack, heap and memory mapping base addresses
- OS, compiler, linker: Exectuables and libraries
  - Position-independent or relocatable code

Compiler & Linker support

# Randomization of layout

Use source of randomness to randomize within ranges



# ASLR – implementation issues

In general: Usage of fixed addresses not allowed

Code should be position-independent or relocatable

Linux/ELF:

- PIC & PIE supported, libraries all PIC, executables sometimes still prelinked

Windows/PE:

- No PIC support! But libraries/executables relocatable!

x86 32bit PIE/PIC slower, no IP relative data addressing
Relocated PE images not sharable between processes

# ASLR effectiveness

Enough entropy?

- Base range size
- Alignment constraints
- Address width (64bit is better than 32bit)

Randomization strategy

Per process, system-wide per boot

Source of randomness

**ASLR only effective without exceptions**

# Bypassing ASLR

Low entropy

Brute-force addresses (multiple attempts required)

Memory leaks (information disclosure)

Leak addresses to derive base addresses
Construct and enforce a leak by memory corruption

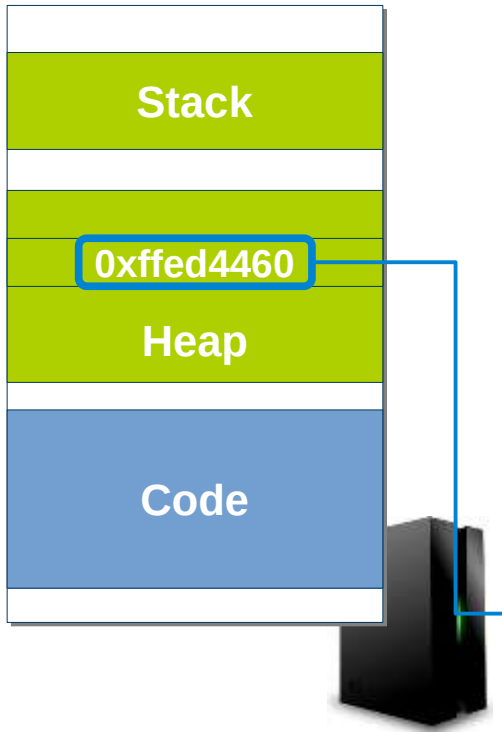
Application and vulnerability specific attacks

Force predictable memory state
- Heap-spraying / Heap massaging

Pointer inference
- Use a side-channel

Avoid using exact addresses
- Only corrupt least significant bytes i.e. offsets

# Memory leak




Runs web server software with memory leak and exploitable memory corruption vulnerability

Victim | Attacker

GET /index.html HTTP/1.1
Host: www.vulnsite.com
Keep-Alive: 300
Connection: keep-alive
Cookie: CID=r2t5uvjq43
5r4q7ib3vtdjq120f83jf8
... <binary data>

**1** Trigger memory leak

HTTP/1.1 200 OK
Date: Fri, 12 Mar 2014 23:59:00 GMT
Content-Type: text/html
Content-Length: 1354
Allow: GET,POST,OPTIONS,HEAD
Cache-Control: public,max-age=30
<binary data> 0x41416232
0xffed4460 0x77ff2332 0x...

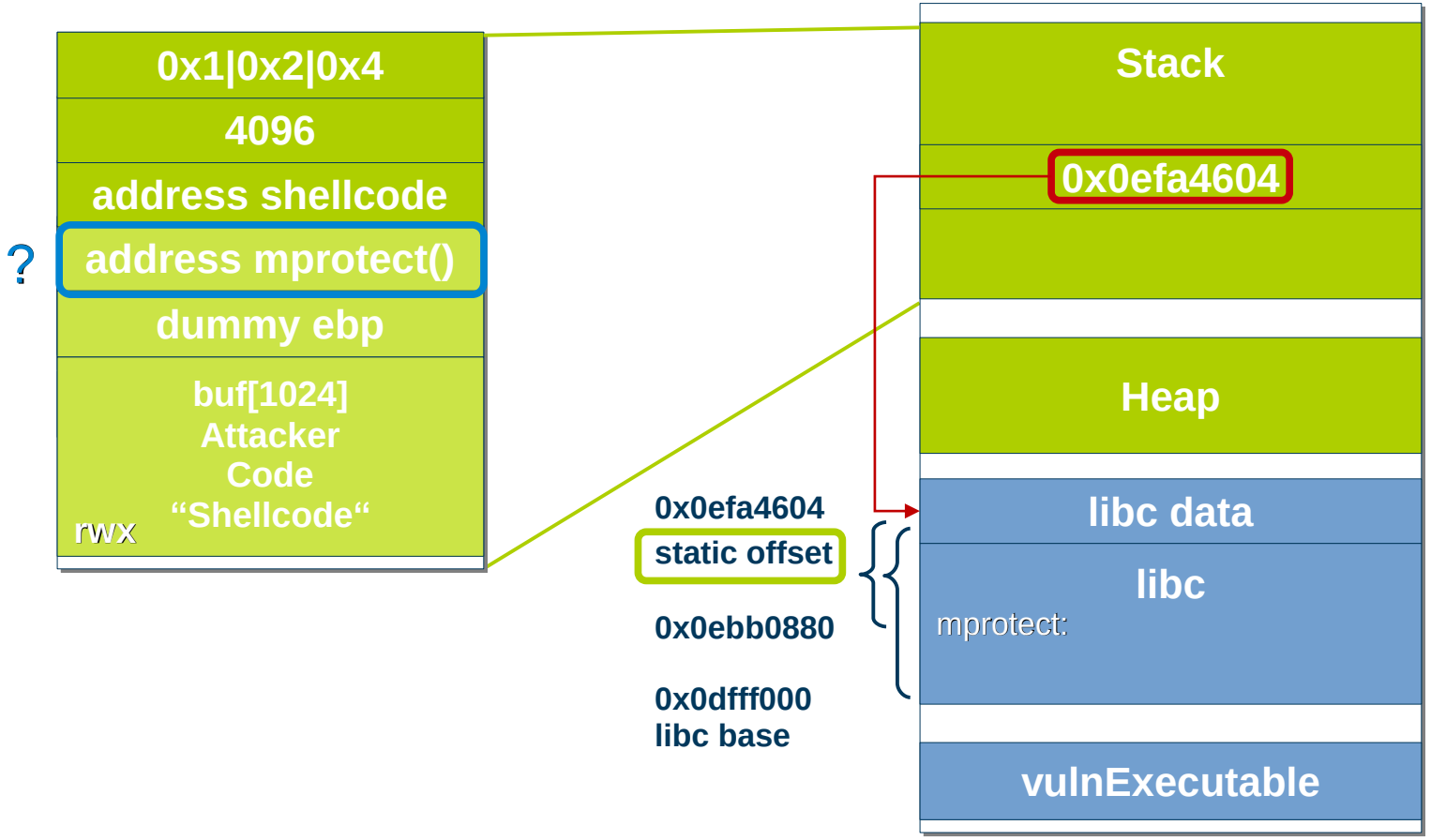
<html>
<head>...

**2** Parse response for leaked memory and **construct exploit**

GET /index.html HTTP/1.1
Host: www.vulnsite.com
Keep-Alive: 300
Connection: keep-alive
Cookie: CID=r2t5uvjq43
5r4q7ib3vtdjq120f83jf8
... <binary data>

**3** Exploit memory error with tailored exploit

# Memory leak




**mprotect = leaked pointer – static offset**

**0x0ebb0880 = 0x0efa4604 - 0x003f3d84**

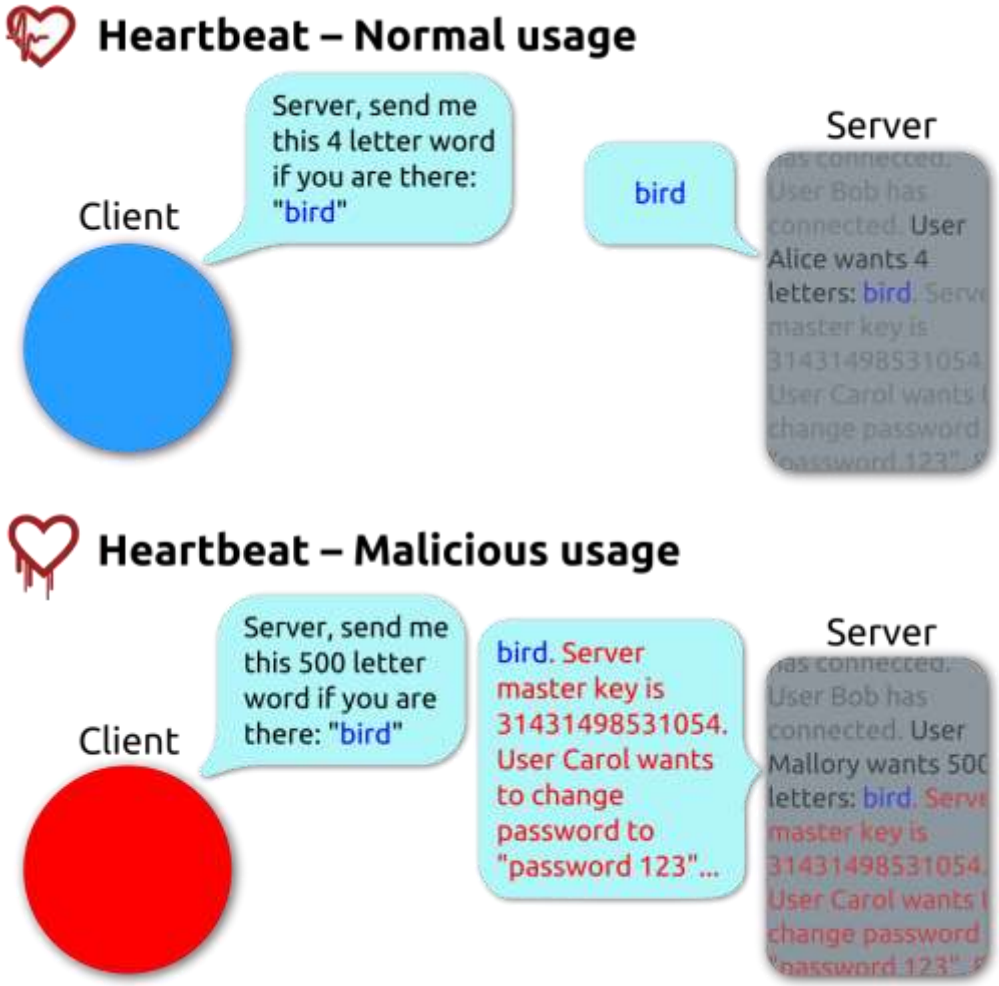
# Memory leak – format string bug

```c
void memLeak() {
        char buf[256];
        scanf("%s", buf);
        printf(buf);
}
```

```c
void main() {
        printf("echo> ");
        memLeak();
        printf("\n");
        return;
}
```

```
shell:~$ gcc -o memleak memleak.c
memleak.c: In function ‘memLeak’:
memleak.c:9:2: warning: format not a string literal and no format arguments [-Wformat-security]
shell:~$ ./memleak
echo> hi
hi
shell:~$ ./memleak
echo> %llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx
1,7fabf517cac0,a,ffffffff,0,6c6c252c786c6c25,252c786c6c252c78,786c6c252c786c6c
shell:~$
```

# Heartbleed – CVE-2014-0160 - OpenSSL




http://en.wikipedia.org/wiki/Heartbleed
Simplified Heartbleed explanation by FenixFeather licensed under CC

# Constructed arbitrary read CVE-2011-2371 – Firefox vulnerability

```
232 // this returns a string at absolute address addr_to_disclose with
233 // length numberbytes
234 function arbitrary_leak_string(addr_to_disclose, numberBytes) {
235
236     array_before_slot_of_infoDisclosure[126] = addr_of_infoDisclosure_slot-8-8-512;
237     // NOTE: addr_of_infoDisclosure_slot-8-8 only works if the heap massage worked out i.e.
238     // the infoDisclosure reallocated slot is adjacent to the other allocations of 512 bytes
239     // i.e. attacker controlled memory, in our case we have something in-between which requires
240     // us to adapt this value so that the str struct pointer points correctly to the flags & length
241     // part of the string structure
242     // SEEMS THAT THERE IS EXACTLY ONE 512 bytes BLOCK IN BETWEEN! -512 required!
243     array_before_slot_of_infoDisclosure[127] = 0xffff0005; // type string
244     // ptr to string structure [124]
245
246     var tmpLen = numberBytes;
247     if(tmpLen % 2 != 0) tmpLen += 1; // multiple of two
248     tmpLen /= 2; // unicode is half of the length
249
250     array_before_slot_of_infoDisclosure[124] = (tmpLen << 4 | 4); // length of unicode string
251     array_before_slot_of_infoDisclosure[125] = addr_to_disclose;
252
```

# DEP & ASLR

DEP & ASLR are generic defenses

Exploitation becomes harder for all vulnerability classes & attack techniques

Together quite effective

- If implemented correctly

Injecting code and corrupting pointers with exact addresses is in general desirable for attackers

# DEP & ASLR are not enough

A determined attacker will use **code-reuse** techniques and **memory leaks** to bypass DEP & ASLR.

# Additional protections

Raise exploit development costs with additional protections

- The more line of defenses, the better!
- Implement protections against specific vulnerability classes and exploitation techniques

# Additional protections

May require **source code changes** (annotations) and/or **recompilation** of the application

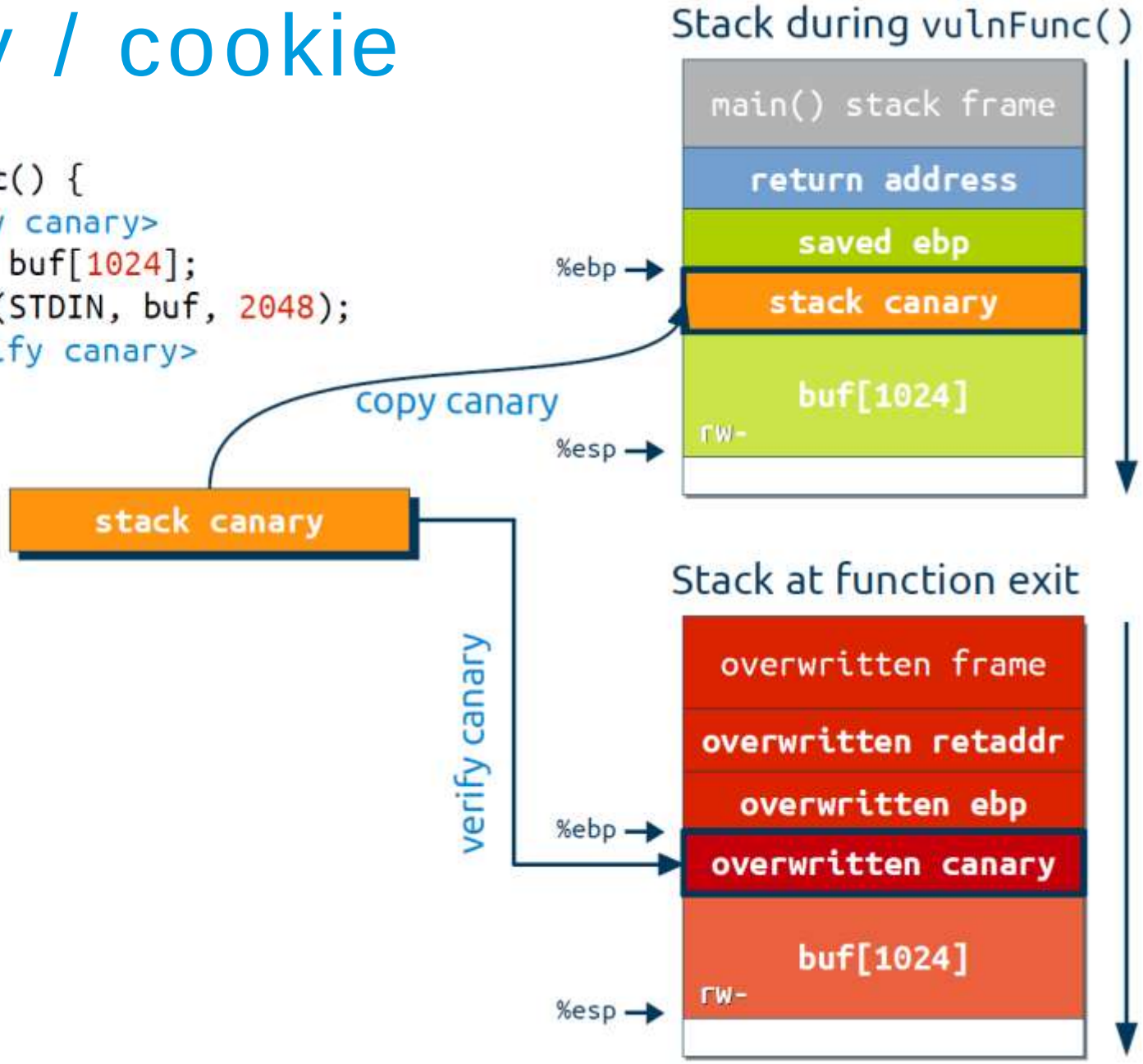
## Compile-time and checks at run-time

Compiler support

- Stack canaries / cookies
- Pointer obfuscation
- /GS
- /SAFESEH (link-time, provide list of valid handlers)
- SEHOP (run-time, walk down SEH chain i.e. integrity check)
- Virtual Table Verification (VTV) & vtguard
- Control-Flow Guard

# Stack canary / cookie

```
void vulnFunc() {
        <copy canary>
        char buf[1024];
        read(STDIN, buf, 2048);
        <verify canary>
}
```

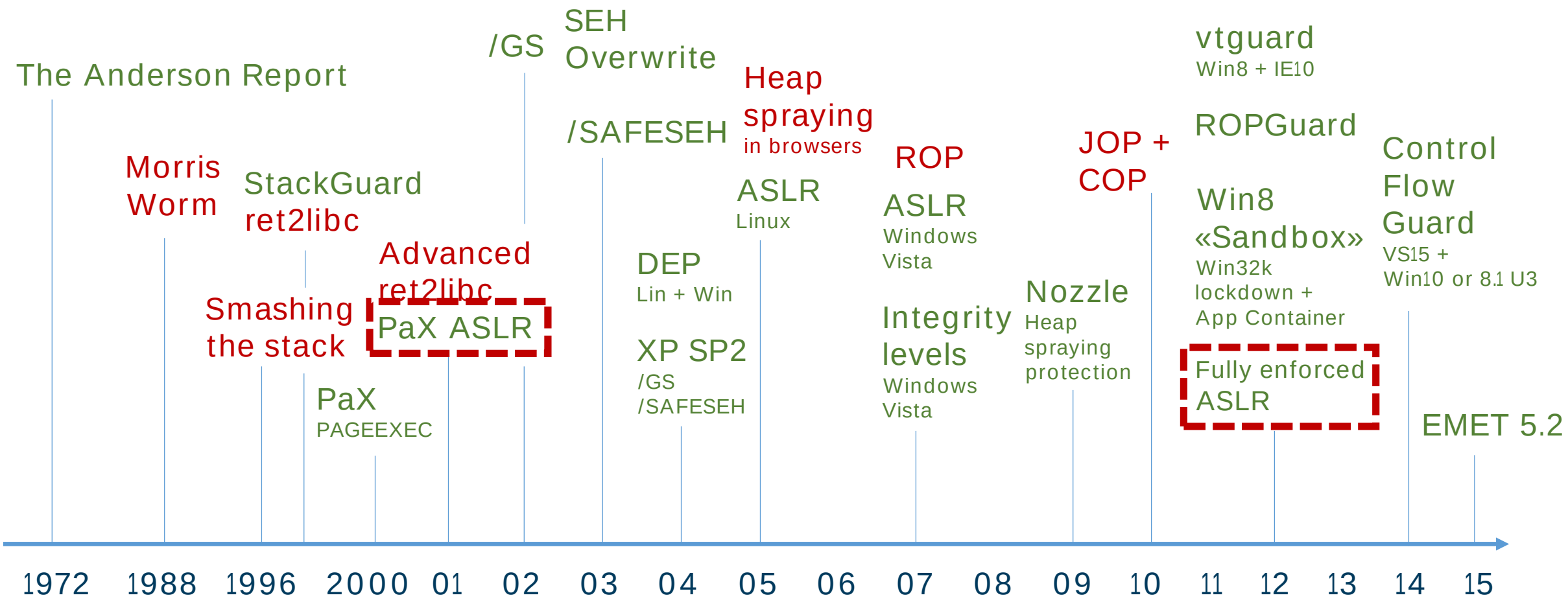


# Mitigations deployment delay

- Deployment of exploit mitigations requires time
- Source or binary compatibility risk
- Support from multiple sides required

Attackers have time to adapt

# Exploit mitigations since the 90s

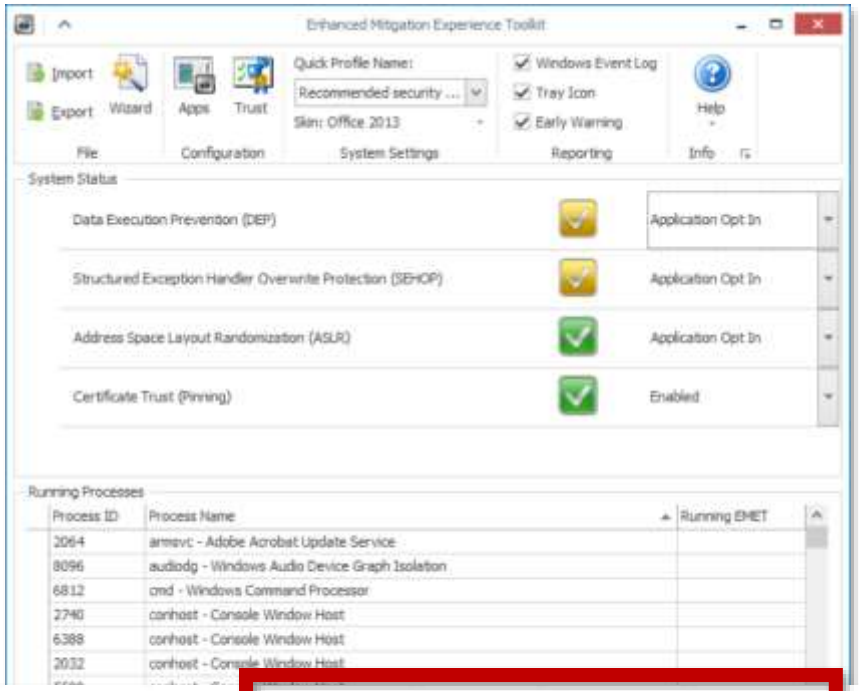


# Modern mitigations

- ROP mitigations (EMET)
- Control-Flow Integrity
  - Control-Flow Guard (CFG)

# ROP mitigations (EMET)

- Run-time checks
- Detect ongoing ROP attacks
- Hook into sensitive APIs




| App Name | DEP | SEHOP | NullPage | HeapSpray | EAF | EAF+ | MandatoryASLR | BottomUpASLR | LoadLib | MemProt | Caller | SimExecFlow | StackPivot | SR |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Acrobat.exe | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| AcroRd32.exe | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | | | |
| EXCEL.EXE | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| iexplore.exe | | | | | | | | | | | | | | |
| INFOPATH.EXE | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |

# Control-flow integrity (CFI)

- Original publication in 2005
  «Control-Flow Integrity – Principles, Implementations, and Applications» Abadi, Budiu, Erlingsson, Ligatti, CCS'05

- Many CFI implementations proposed since then
  - Compiler-based, binary-only (static rewriting)

- Check indirect control-flow transfers and limit the set of allowed targets

# Control-flow integrity (CFI)

- CFI approaches
  XFI (2006), HyperSafe (2010), CFIMon (2012), MoCFI (2012), BinCFI (2013), KCoFI (2014), MCFI (2014)

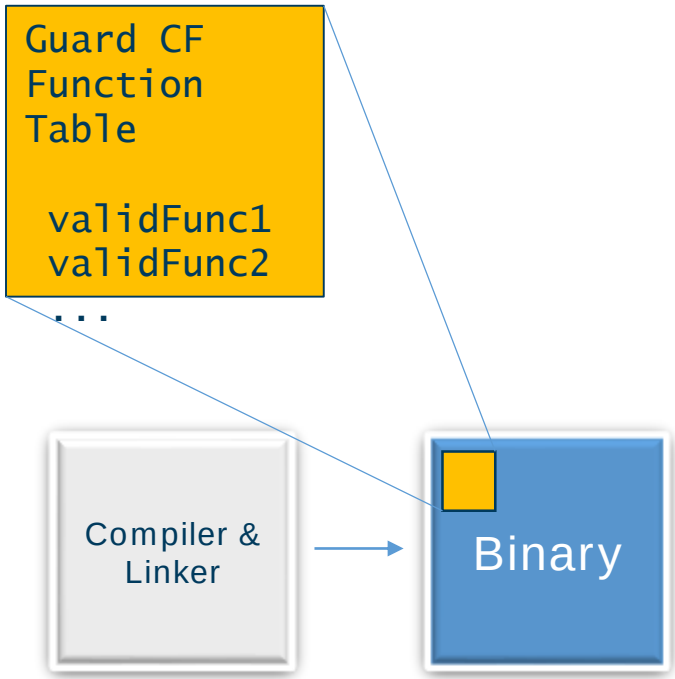
- "Fine-Grained Control-Flow Integrity through Binary Hardening", DIMVA 2015
  Mathias Payer, Antonio Barresi, Thomas R. Gross

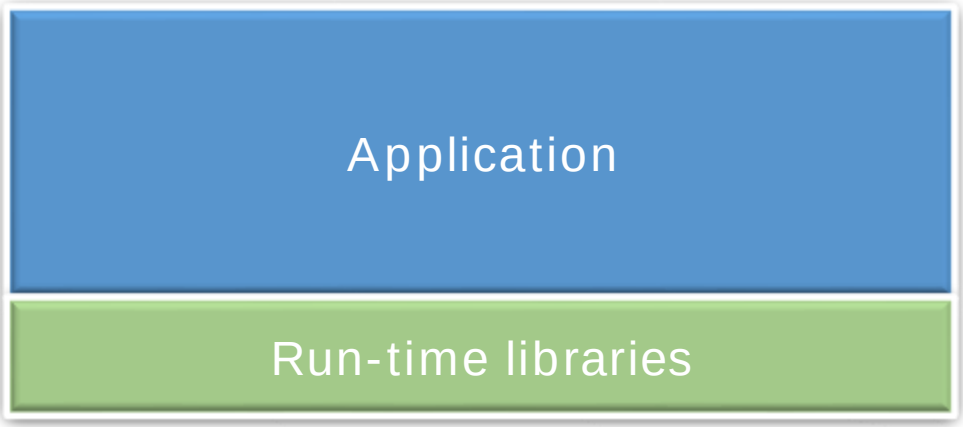
- "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity", USENIX 2015
  Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, Thomas R. Gross

# Control-flow guard (CFG)

- First adoption of a practical Control-Flow Integrity (CFI) implementation
- Requires recompilation and OS support
  - VS15 + Windows 10 or 8.1 U3
- Restricts indirect call targets to a static global set of locations

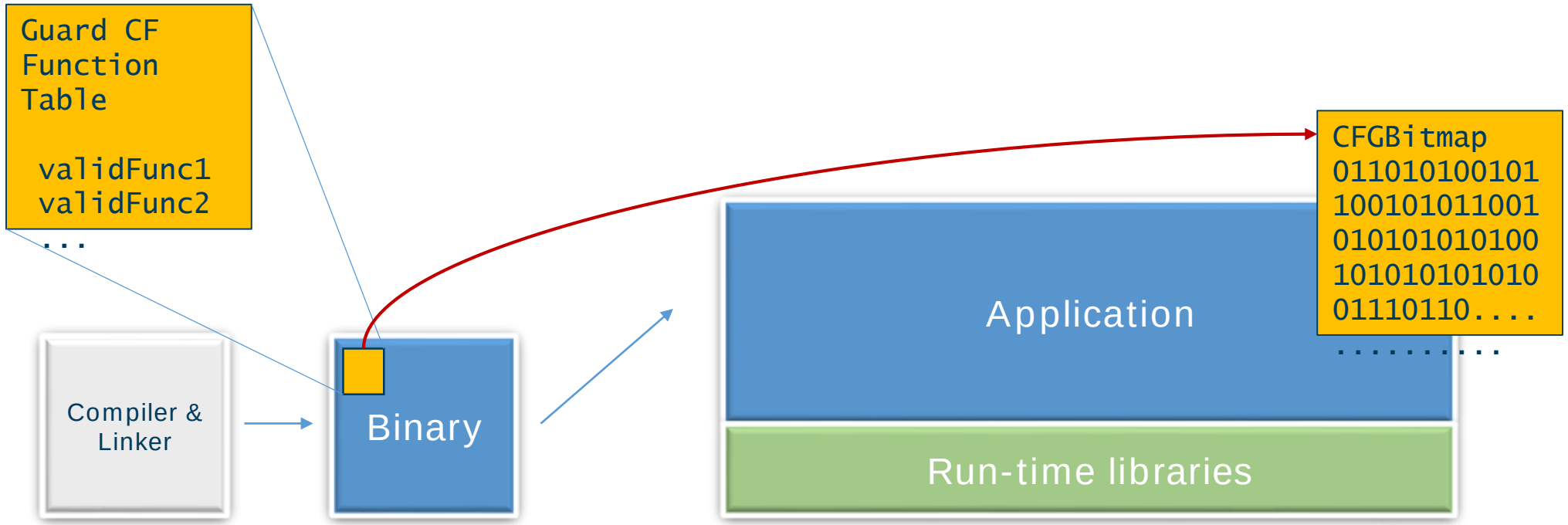
# Control-flow guard (CFG)



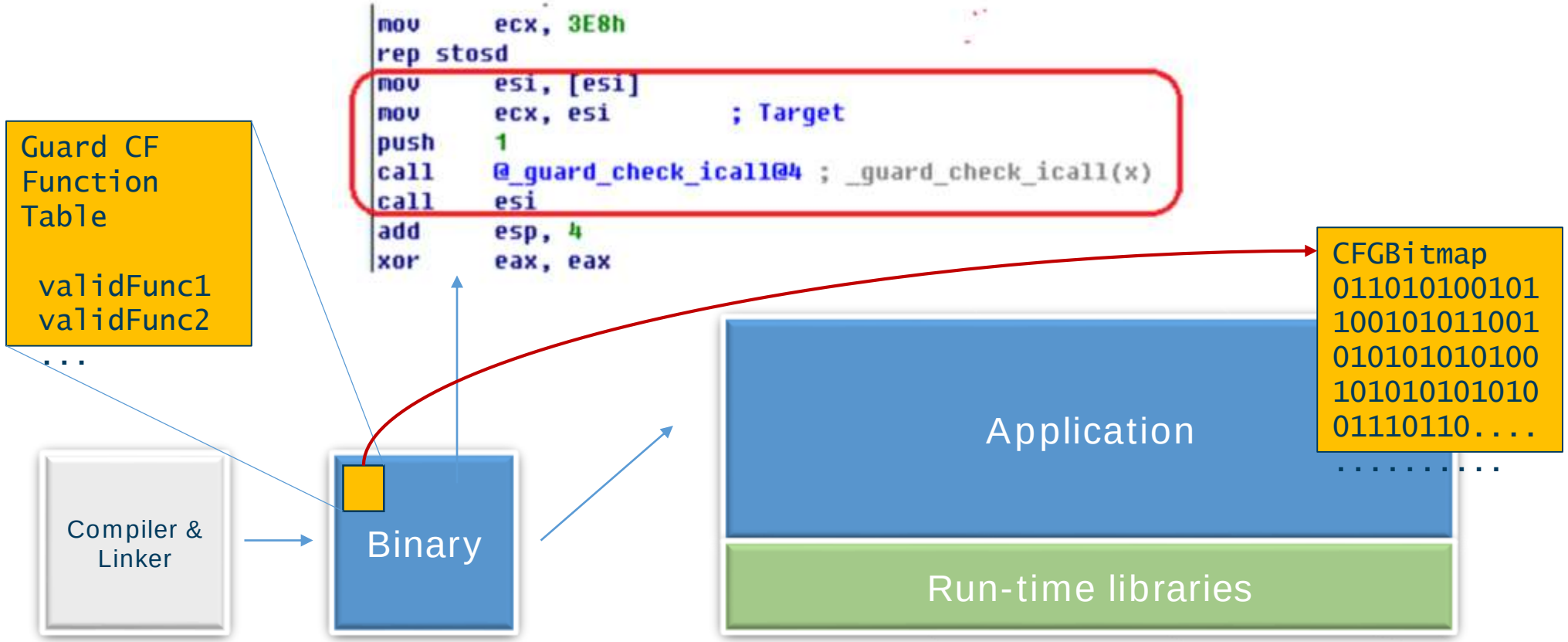

http://sjc1-te-ftp.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf

# Control-flow guard (CFG)



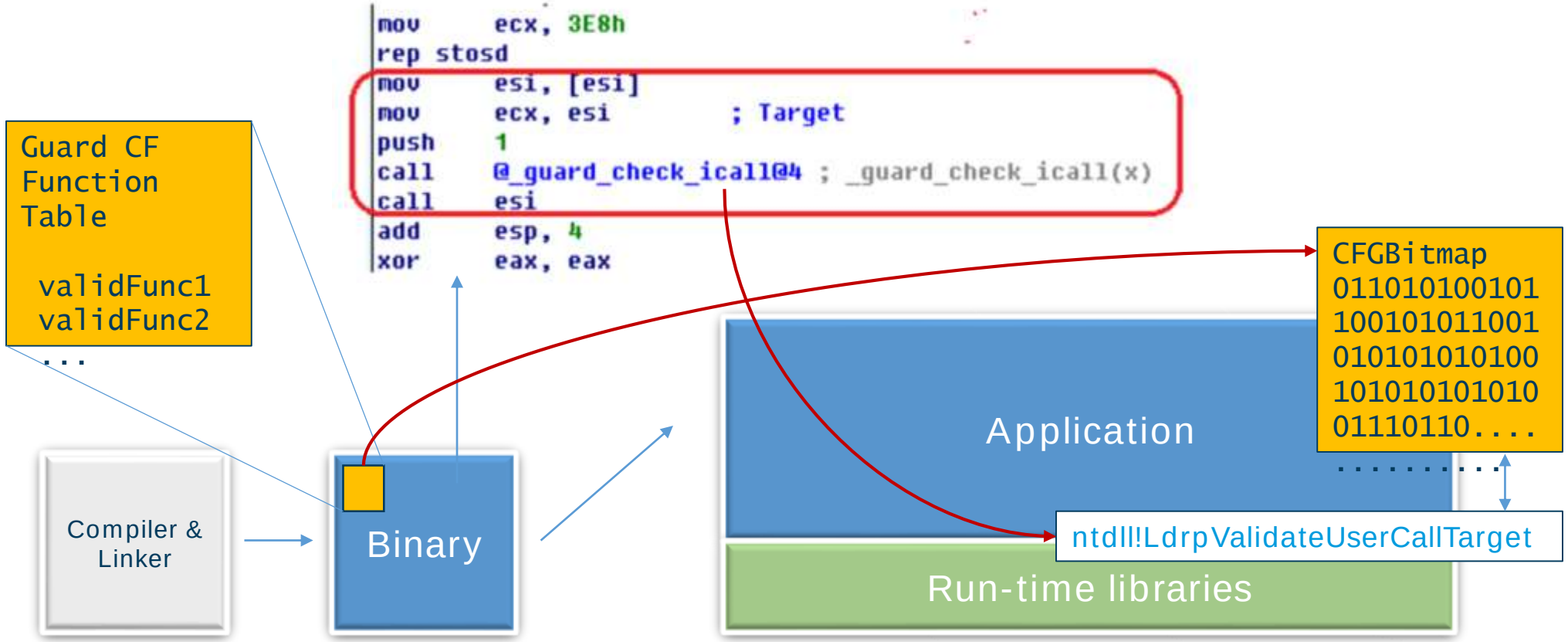

http://sjc1-te-ftp.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf

# Control-flow guard (CFG)




http://sjc1-te-ftp.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf

# Control-flow guard (CFG)




http://sjc1-te-ftp.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf

# Future?

- Hardening techniques make exploit development harder and increase the costs of an attack
- But there is no magic bullet
- Attacker's still leverage memory leaks and code-reuse techniques to bypass all mitigations (ret2libc is 1997)

# Pwn2own 2016

## Pwn2Own 2016: Chrome, Edge, and Safari hacked, $460,000 awarded in total

EMIL PROTALINSKI MARCH 18, 2016 12:21 PM

TAGS: ADOBE, ADOBE FLASH, APPLE, APPLE SAFARI, CHROME, EDGE, FLASH, GOOGLE, GOOGLE CHROME, MICROSOFT, MICROSOFT EDGE, PWN2OWN, SAFARI, TR MICRO


**Pwn2Own 2016 in Numbers:**

Total prizes awarded:

- Master of Pwn Points: 98
- Cash: US$ 460,000

Number of Attempts:

- Fully Successful: 7 (64%)
- Partially Successful: 1 (9%)
- Failed: 3 (27%)

Number of Successful Attempts Against:

- Apple Safari: 3/3 (100% Success)
- Microsoft Edge: 2 /2 (100% Success)
- Adobe Flash: 4/5 (75% Success)
- Google Chrome: .5/2 (25% Success) NOTE: The actual vulnerability in Google Chrome had already been independently reported to Google; this is counted a partial success)

Percentage of Successful or Partially Successful attacks that achieved SYSTEM or root privilege: 100%

Contestant Success Standings:

- Tencent Security Team Sniper (KeenLab and PC Manager): 3/3 (100% Success)
- 360Vulcan Team: 1.5/2 (75% Success)
- JungHoon Lee (lokihardt): 2/3 (66% Success)
- Tencent Security Team Shield (PC Manager and KeenLab): 1/2 (50% Success)
- Tencent Xuanwu Lab: 0/1 (0% Success)

http://venturebeat.com/2016/03/18/pwn2own-2016-chrome-edge-and-safari-hacked-460k-awarded-in-total/
http://blog.trendmicro.com/pwn2own-day-2-event-wrap/

# Conclusion

# Conclusion

- Compilers help mitigate software attacks
- Software hardening increases costs of attacks
- Attackers still find ways to bypass all protections

# Questions ?

antonio.barresi@xorlab.com

# xorlab


```
$ cat xorlab.info

contact@xorlab.com
www.xorlab.com

Universitätsstrasse 6
8092 Zürich

$
```