On Compilers, Memory Errors and Control-Flow Integrity

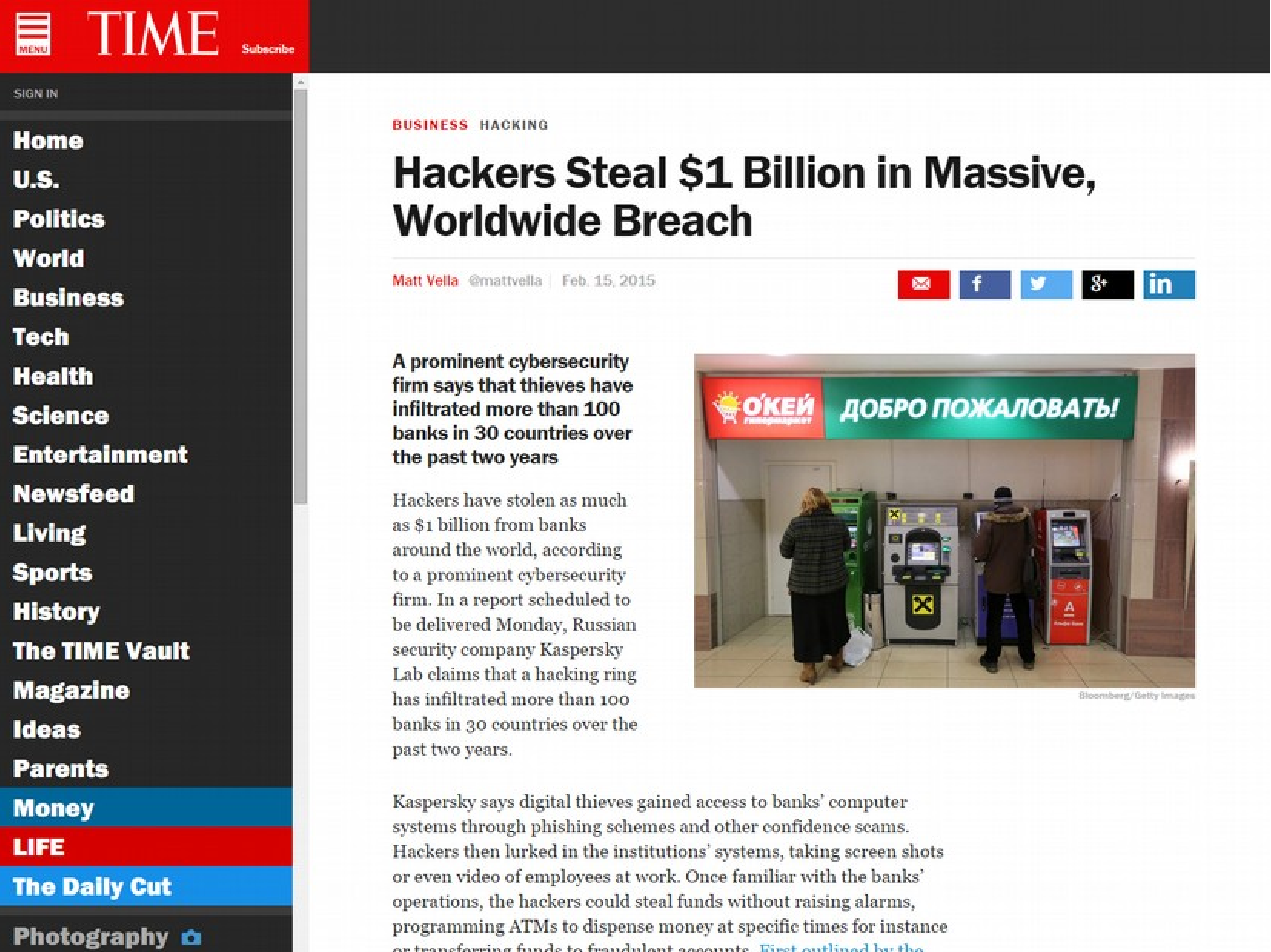
Advanced Compiler Design – SS 2015

Antonio Hüseyin Barresi

Zürich, 27.5.2015

ETH zürich

LST Laboratory for Software Technology



BUSINESS HACKING

Hackers Steal $1 Billion in Massive, Worldwide Breach

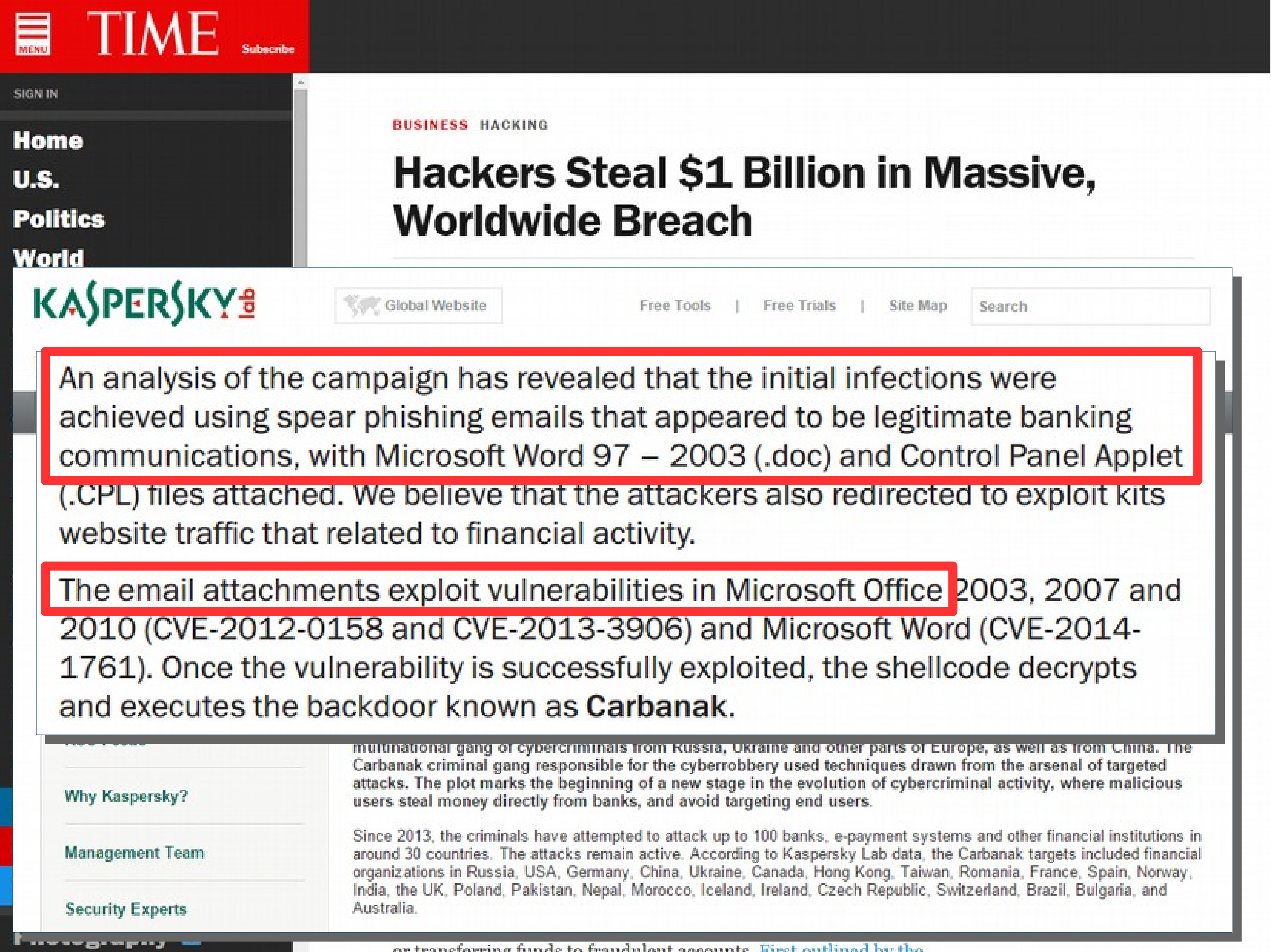
Matt Vella @mattvella | Feb. 15, 2015

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

Bloomberg/Getty Images

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



BUSINESS HACKING

Hackers Steal $1 Billion in Massive, Worldwide Breach

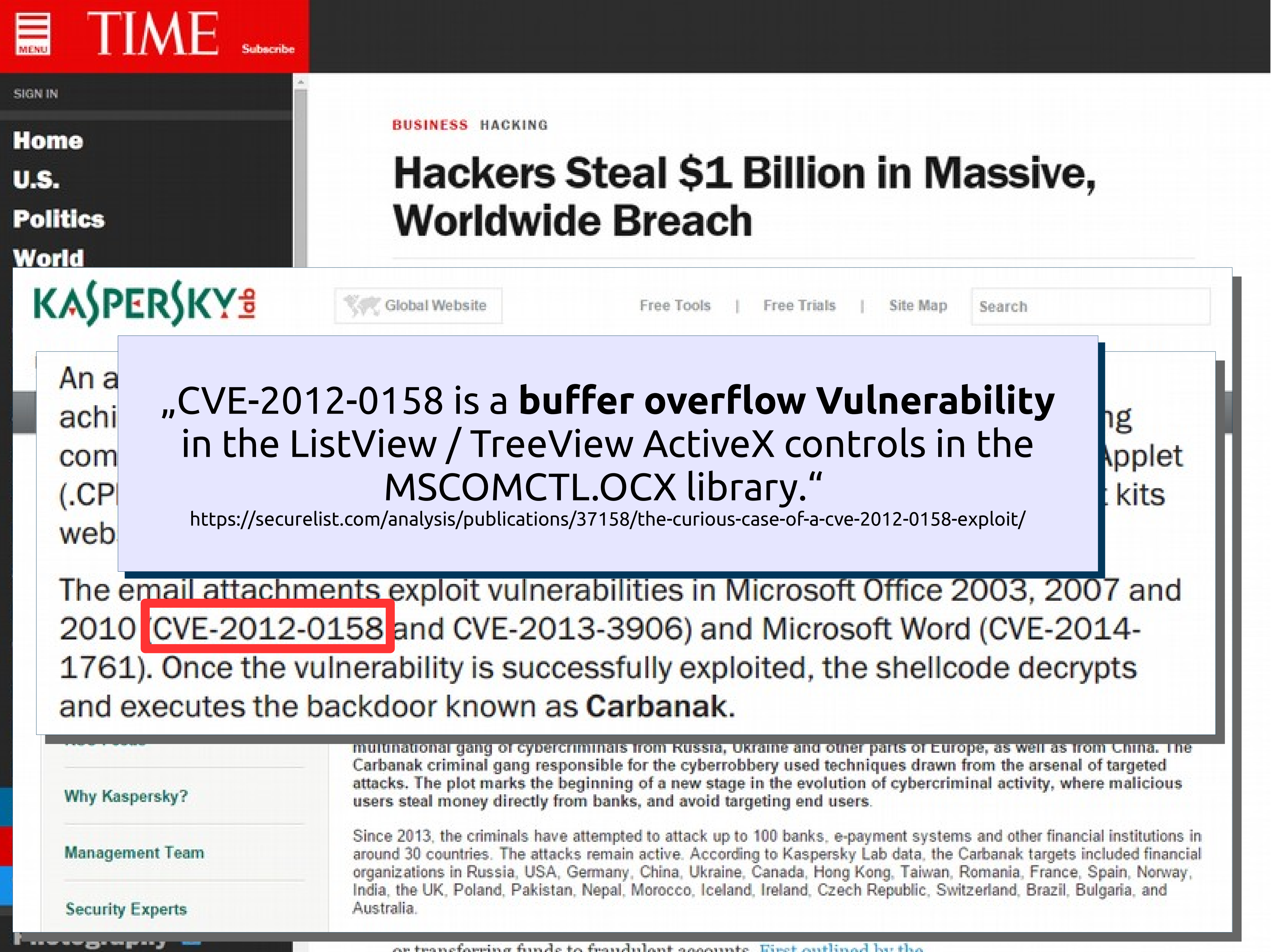
An analysis of the campaign has revealed that the initial infections were achieved using spear phishing emails that appeared to be legitimate banking communications, with Microsoft Word 97 – 2003 (.doc) and Control Panel Applet (.CPL) files attached. We believe that the attackers also redirected to exploit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

multinational gang of cybercriminals from Russia, Ukraine and other parts of Europe, as well as from China. The Carbanak criminal gang responsible for the cyberrobbery used techniques drawn from the arsenal of targeted attacks. The plot marks the beginning of a new stage in the evolution of cybercriminal activity, where malicious users steal money directly from banks, and avoid targeting end users.

Since 2013, the criminals have attempted to attack up to 100 banks, e-payment systems and other financial institutions in around 30 countries. The attacks remain active. According to Kaspersky Lab data, the Carbanak targets included financial organizations in Russia, USA, Germany, China, Ukraine, Canada, Hong Kong, Taiwan, Romania, France, Spain, Norway, India, the UK, Poland, Pakistan, Nepal, Morocco, Iceland, Ireland, Czech Republic, Switzerland, Brazil, Bulgaria, and Australia.

or transferring funds to fraudulent accounts. First outlined by the



BUSINESS HACKING

Hackers Steal $1 Billion in Massive, Worldwide Breach

An a… achi… com… (.CP… web… …ng …Applet …kits

„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/

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**.

multinational gang of cybercriminals from Russia, Ukraine and other parts of Europe, as well as from China. The Carbanak criminal gang responsible for the cyberrobbery used techniques drawn from the arsenal of targeted attacks. The plot marks the beginning of a new stage in the evolution of cybercriminal activity, where malicious users steal money directly from banks, and avoid targeting end users.

Since 2013, the criminals have attempted to attack up to 100 banks, e-payment systems and other financial institutions in around 30 countries. The attacks remain active. According to Kaspersky Lab data, the Carbanak targets included financial organizations in Russia, USA, Germany, China, Ukraine, Canada, Hong Kong, Taiwan, Romania, France, Spain, Norway, India, the UK, Poland, Pakistan, Nepal, Morocco, Iceland, Ireland, Czech Republic, Switzerland, Brazil, Bulgaria, and Australia.

or transferring funds to fraudulent accounts. First outlined by the

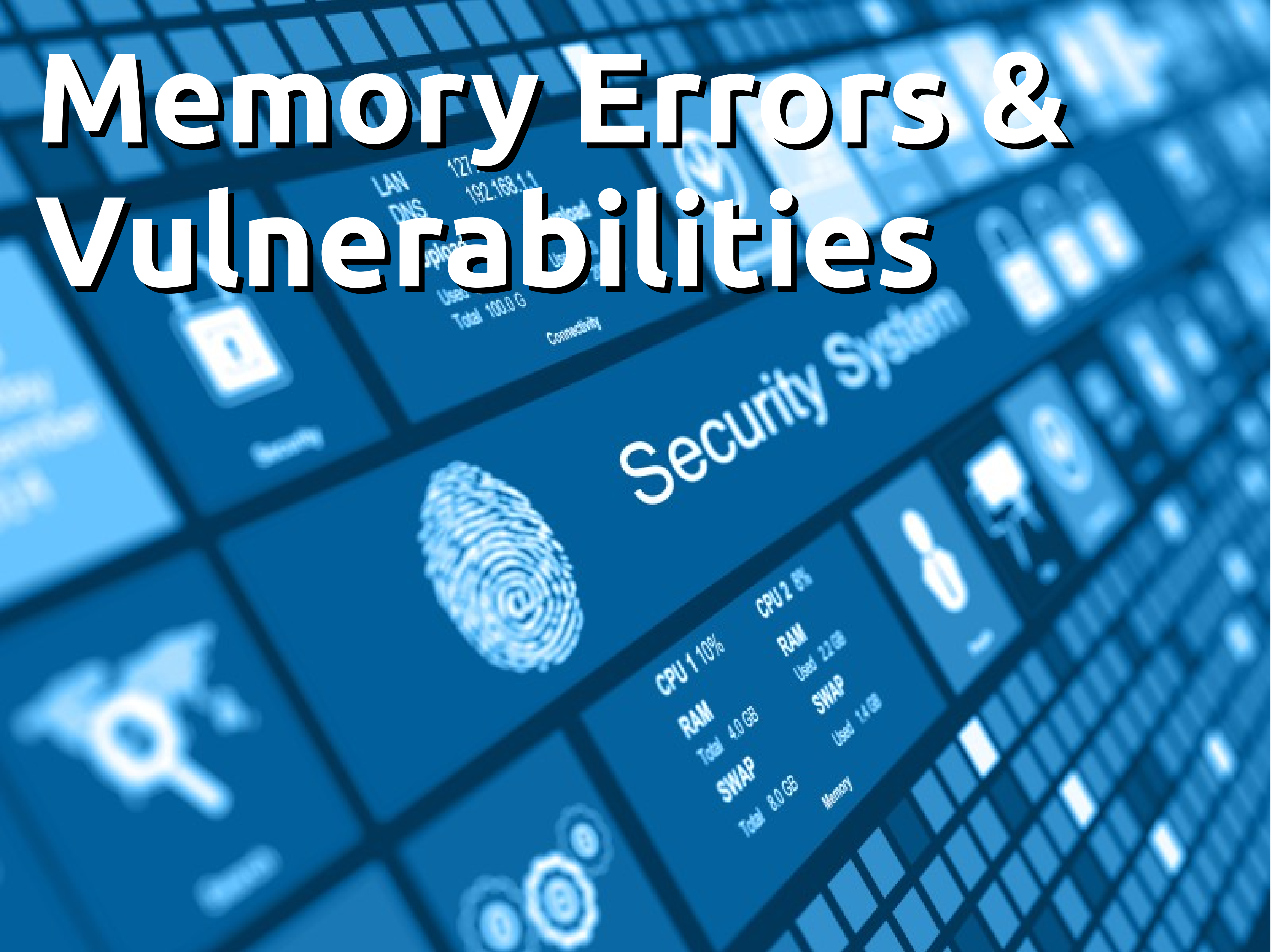
Cyber criminals

How does this relate to compilers?

This talk is about

- Memory errors
- What compilers do about it
- Control-Flow Integrity

Memory Errors & Vulnerabilities



Memory errors & vulnerabilities

- Come in various forms
- Allow attackers to corrupt memory in a more or less controllable way
  - Worst case: attackers gain arbitrary code execution
- Exist in programs written in “unsafe“ languages that do not enforce memory safety

“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

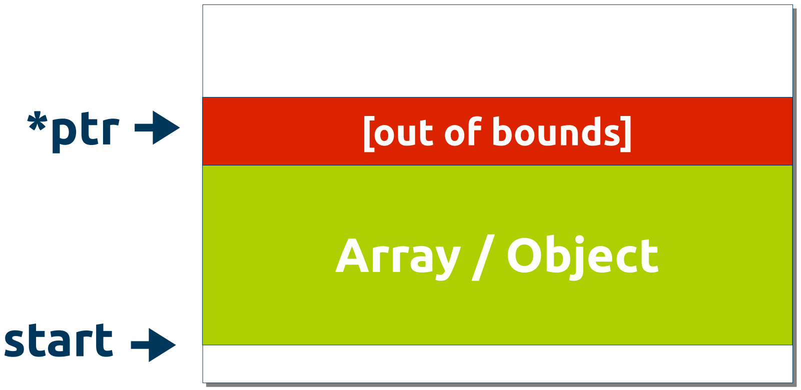
```c
#include <stdio.h>
#include <string.h>

#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>
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...
Segmentation fault (core dumped)
shell:~$
```

Types of memory errors

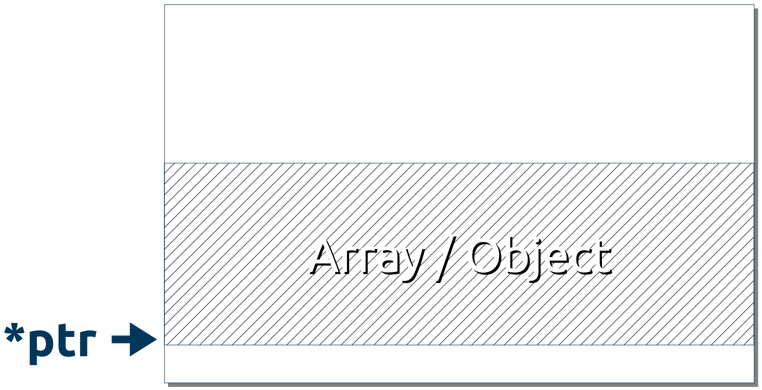
Spatial error



De-reference pointer that is out of bounds

Read or write operation

Temporal error

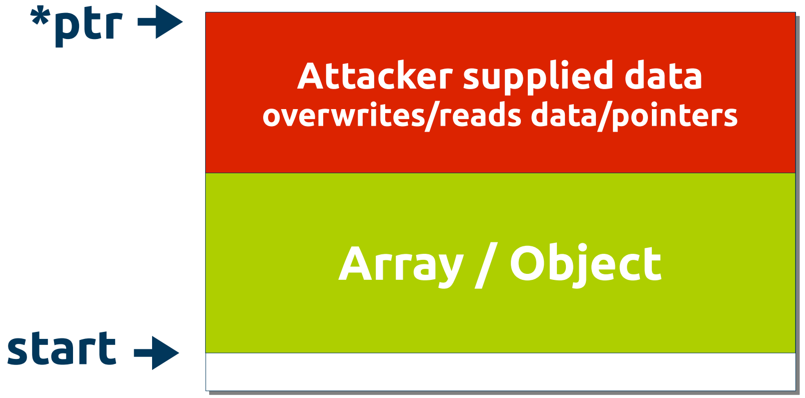


De-reference pointer to freed memory

Read operation

Exploiting memory errors

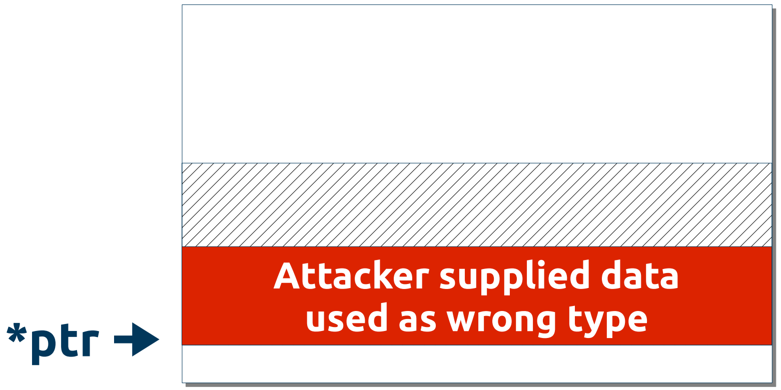
Spatial error



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

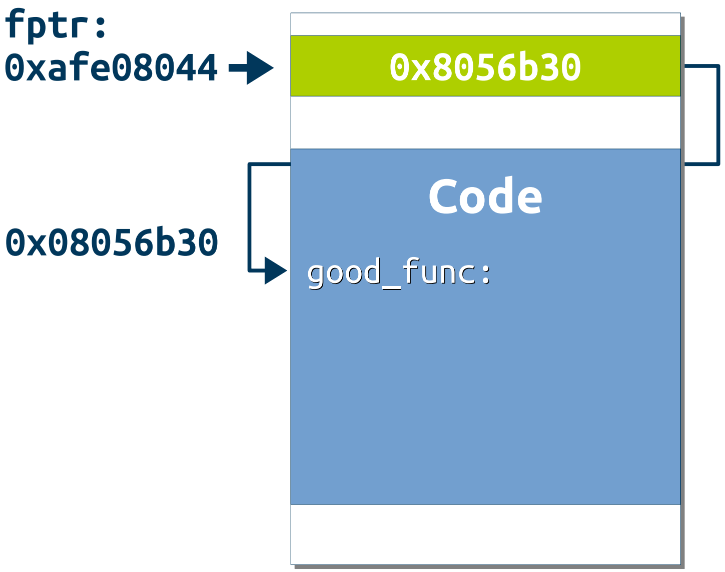
- Overwrite data or pointers
  - That might be used to overwrite data or pointers
  - Function pointers, sensitive data, index values, control-flow sensitive data etc.
- Leak information
  - E.g., corrupt a length field
- Construct attacker primitives
  - Write primitive (write any value to arbitrary address)
  - Read primitive (read from any address)
  - Arbitrary call primitive (call any arbitrary address)

Control-flow hijack attacks

- Most ISAs support indirect branch instructions
  - E.g., x86 “ret“, indirect “jmp“, indirect “call“

fptr is a value in memory at 0xafe08044

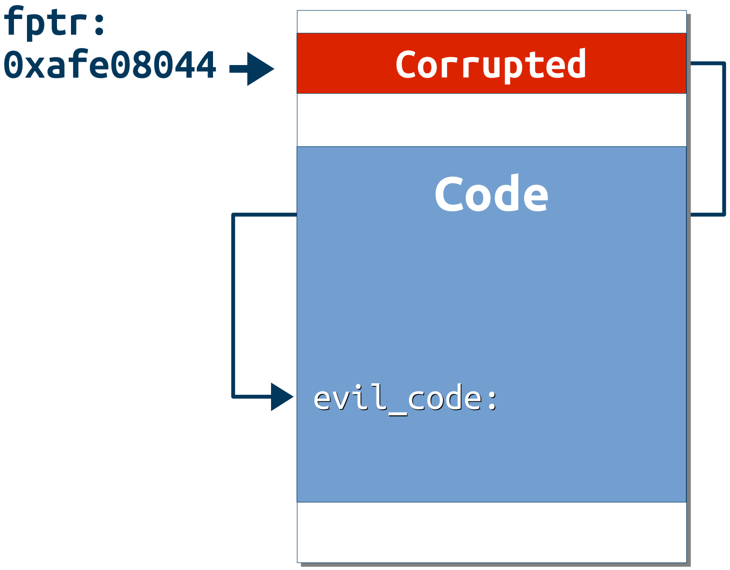
- branch *fptr



Control-flow hijack attacks

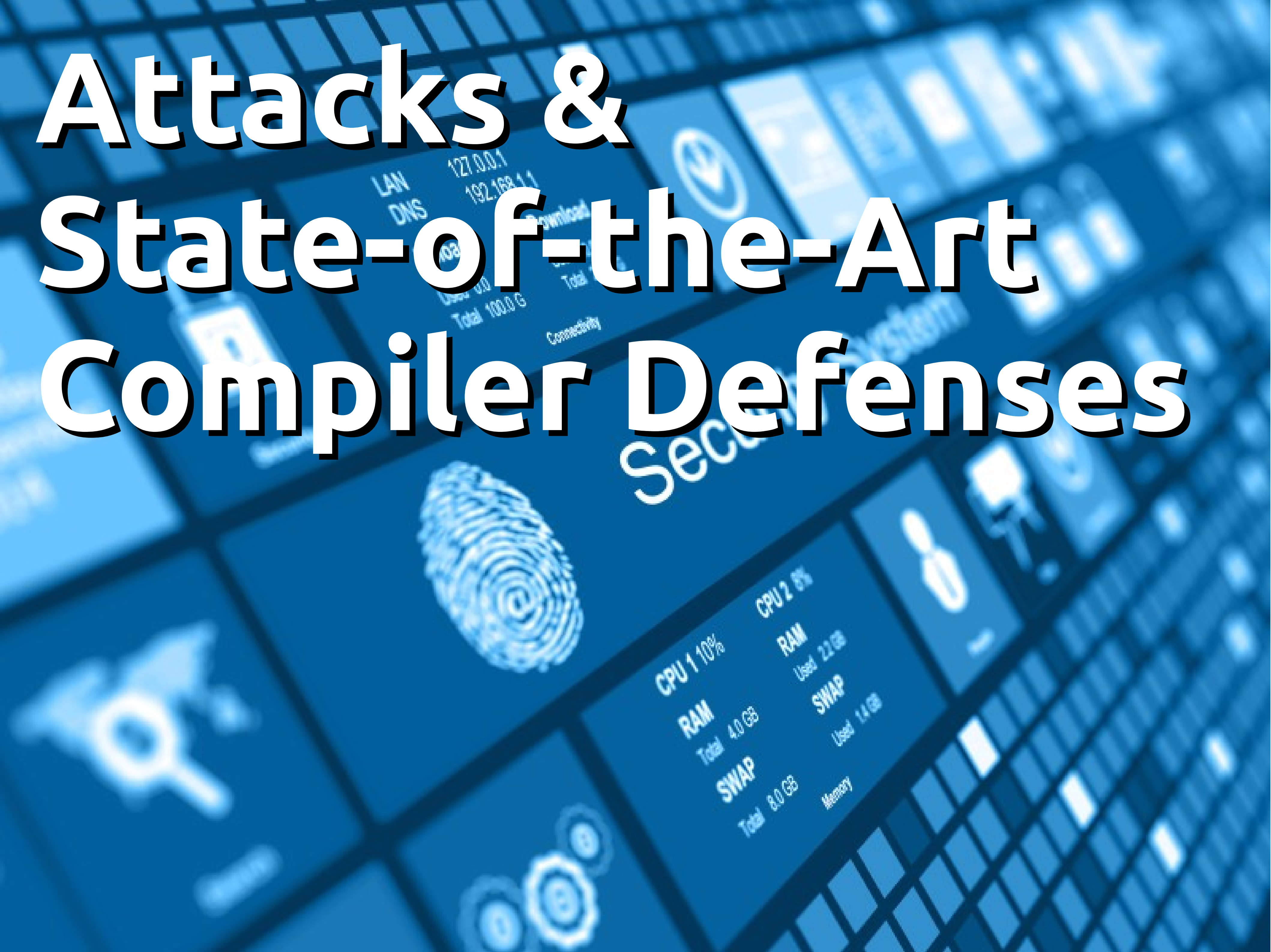
`fptr` is a value in memory at `0xafe08044`

- `branch *fptr`
- `fptr` was corrupted by an attacker

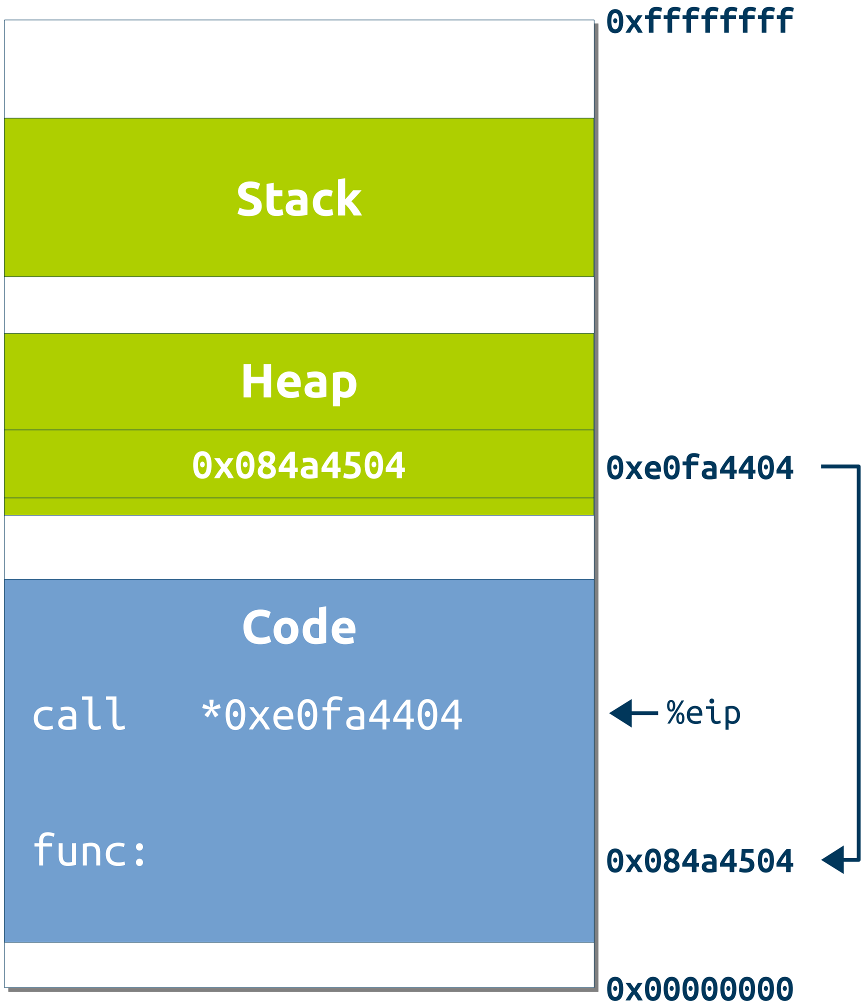


Attacker goal: hijack control-flow to injected machine code or to “evil functions“

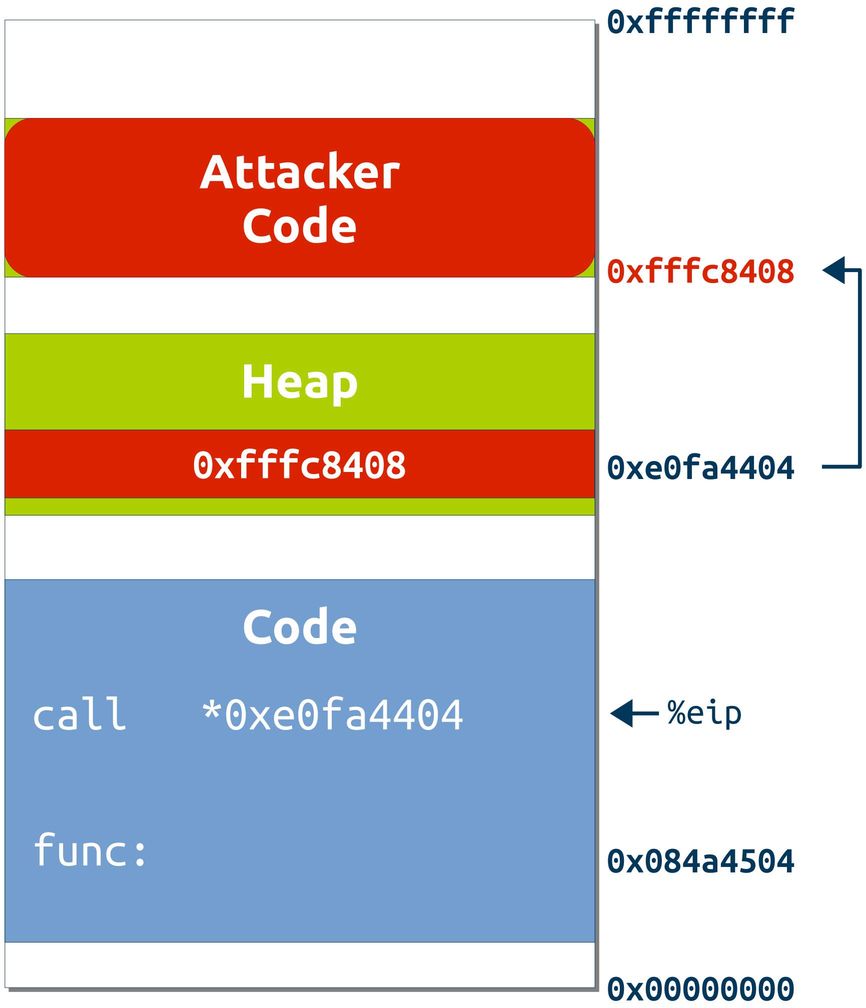
Attacks & State-of-the-Art Compiler Defenses



Control-flow hijack to injected code



Indirect call to func()

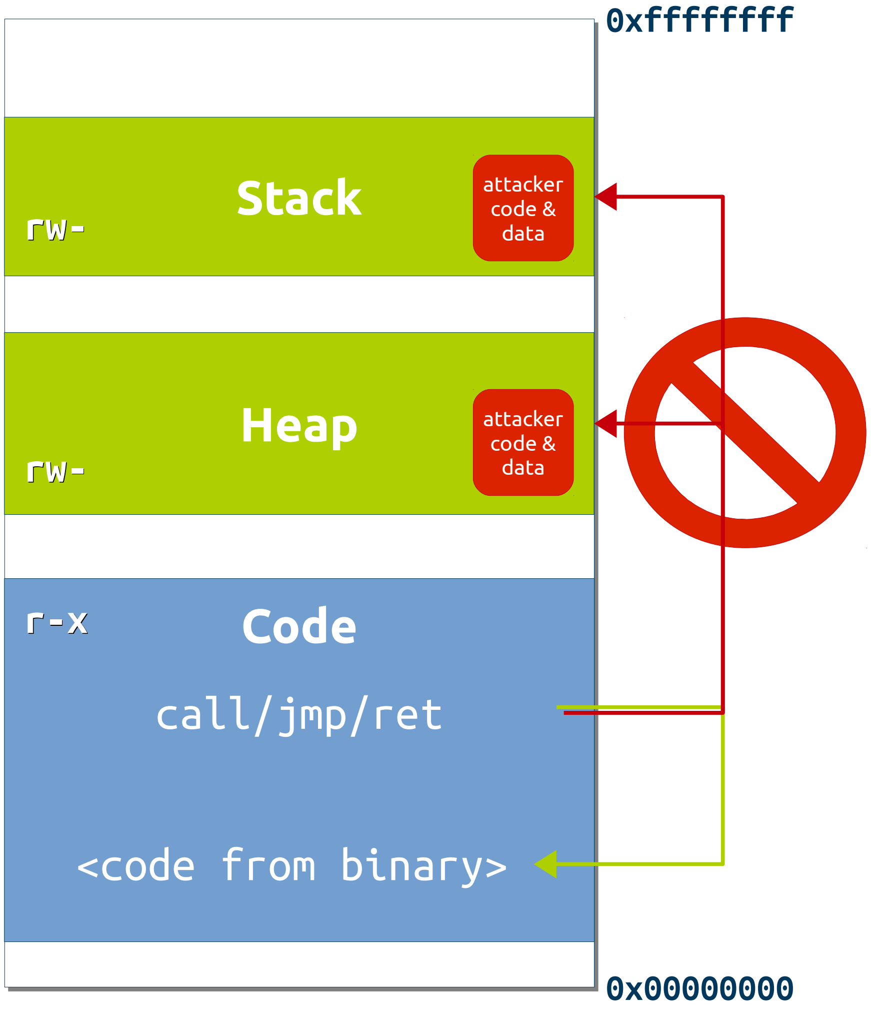


Hijacked indirect call

Non-eXecutable data (NX)

- Make **data** regions **non-executable** (by default)

- Changing protection flags or allocating `rwx` memory still possible (on most systems)
  - Required for JITs



NX / DEP

Compatibility

- 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

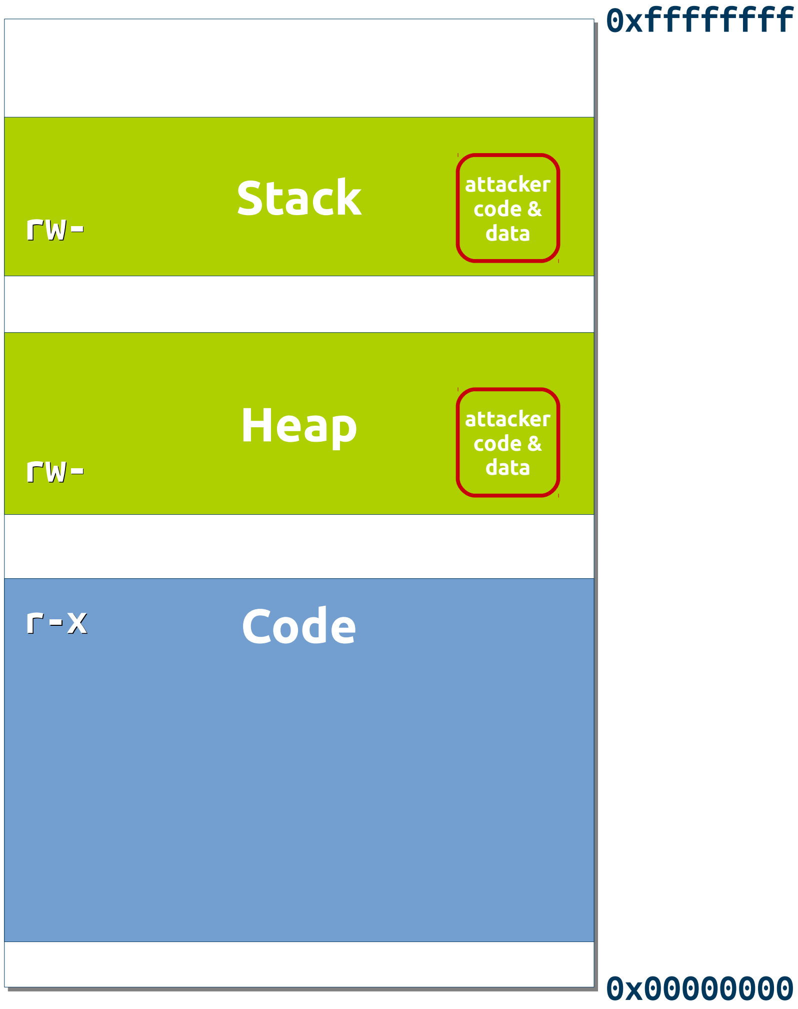
Bypassing NX / DEP

- Only use existing code

  Code-reuse attack

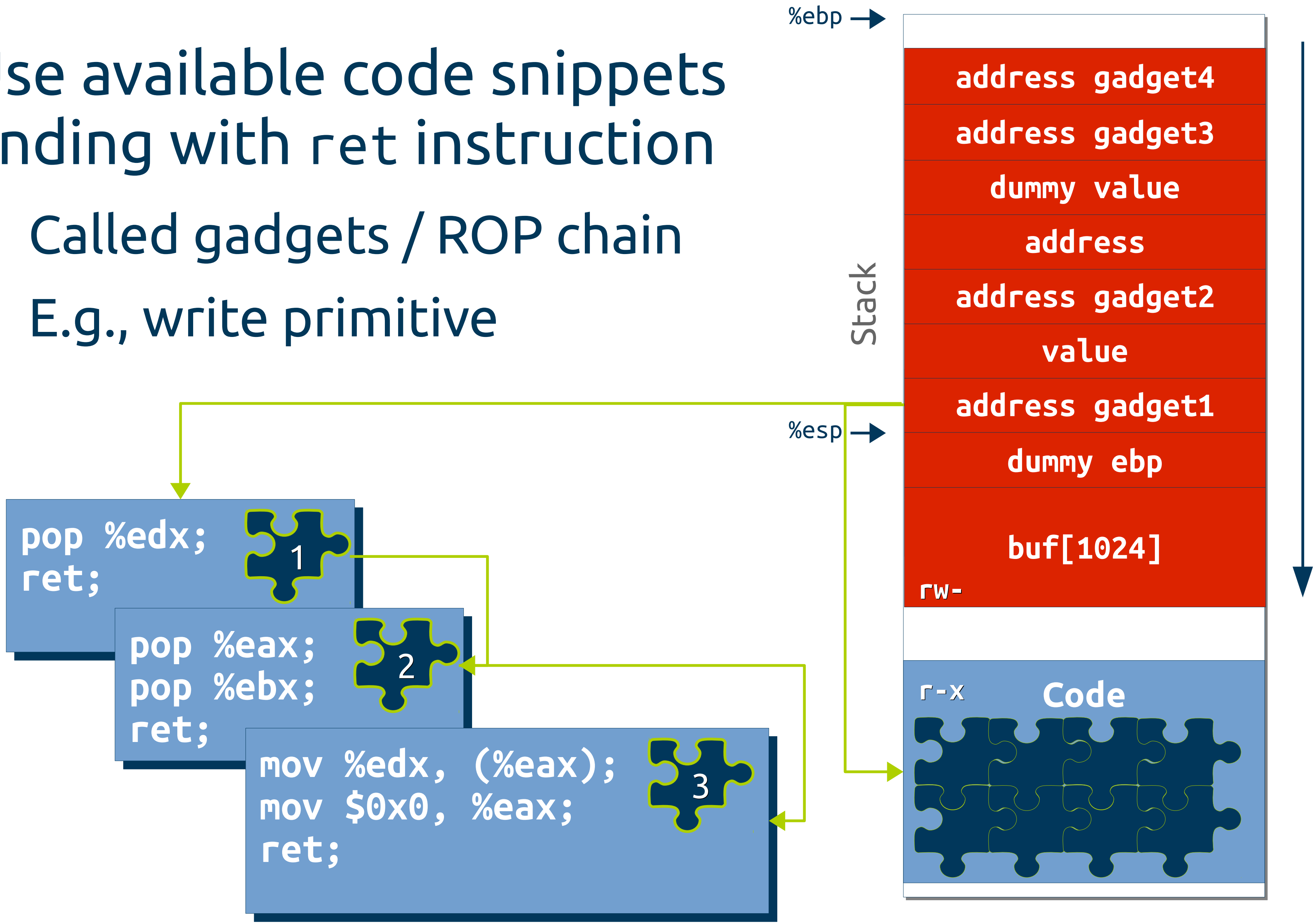
  - ret2libc, ret2bin, ret2*
  - Return-oriented programming (ROP)
  - Jump/Call-oriented programming

- Use code-reuse technique to **change protection flags**
  - Alllocate or make memory executable
    - mprotect/VirtualProtect
    - mmap/VirtualAlloc



Return-oriented programming

- Use available code snippets ending with `ret` instruction
  - Called gadgets / ROP chain
  - E.g., write primitive



%ebp →

address gadget4
address gadget3
dummy value
address
address gadget2
value
address gadget1
dummy ebp
buf[1024]

%esp →

```
pop %edx;
ret;
```

```
pop %eax;
pop %ebx;
ret;
```

```
mov %edx, (%eax);
mov $0x0, %eax;
ret;
```

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 **code (shellcode)**
  - Addresses of **gadgets**
- Sometimes it's enough to know the rough location but **most of the time** attackers need the **exact location**
  - Corrupting only least significant bytes i.e. an offset might work in some special cases (but not in general)

ASLR

Today most Operating Systems implement
Address Space Layout Randomization

- What can be randomized?
  - OS: Stack, heap and memory mapping base addresses
  - OS, compiler, linker: Exectuables and libraries
    - Position-indipendent or relocatable code

COMPILER & LINKER SUPPORT

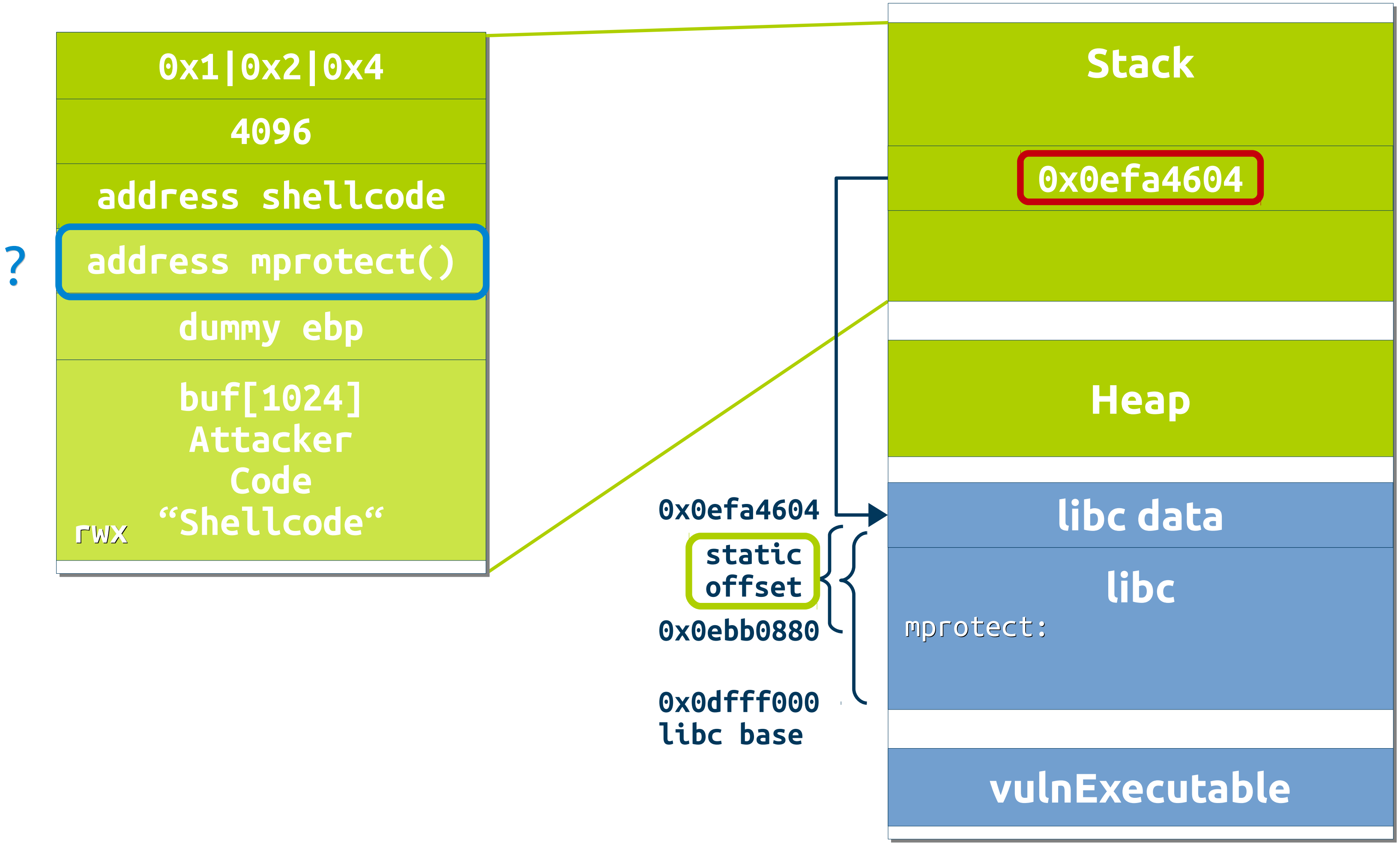
Bypassing ASLR

- Low entropy
  - Brute-force addresses
    (multiple attempts required)

- Memory leaks (information disclosure)
  - Leak addresses to derive base addresses
    - E.g., run-time address pointing into a library
  - Construct and enforce a leak
    by memory corruption

- Application and vulnerability specific attacks

Memory leak



mprotect = leaked pointer - static offset

0x0ebb0880 = 0x0efa4604 - 0x003f3d84

DEP & ASLR

DEP & ASLR are generic defenses

- Exploitation becomes harder for all vulnerability classes & attack techniques
- Together quite effective
  - If implemented correctly and used continuously
- But DEP & ASLR are not enough

Compile-time protections

- Usually require source code changes (annotations) and/or recompilation of the application
  - To add run-time checks

COMPILER SUPPORT

- **Stack canaries / Cookies**
- Pointer obfuscation
- /GS (more than just cookies)
- /SAFESEH (link-time, provide list of valid handlers)
- SEHOP (run-time, walk down SEH chain to final handler before dispatching / integrity check)
- Virtual Table Verification (VTV) & vtguard
- Control-Flow Guard (new in Visual Studio 2015)

Stack canary / cookie

COMPILER SUPPORT

```
void vulnFunc() {

        char buf[1024];
        read(STDIN, buf, 2048);

}
```

Stack canary / cookie

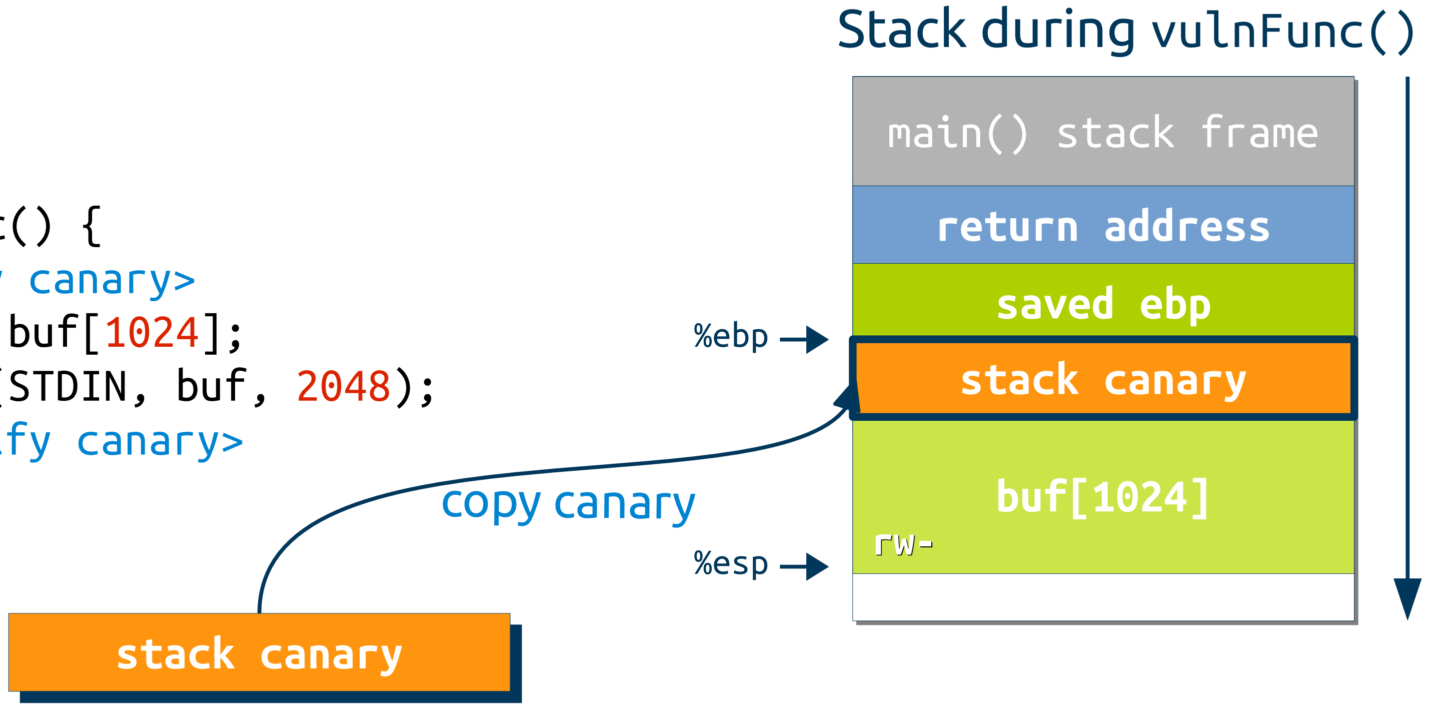
COMPILER SUPPORT

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

Stack canary / cookie

COMPILER SUPPORT

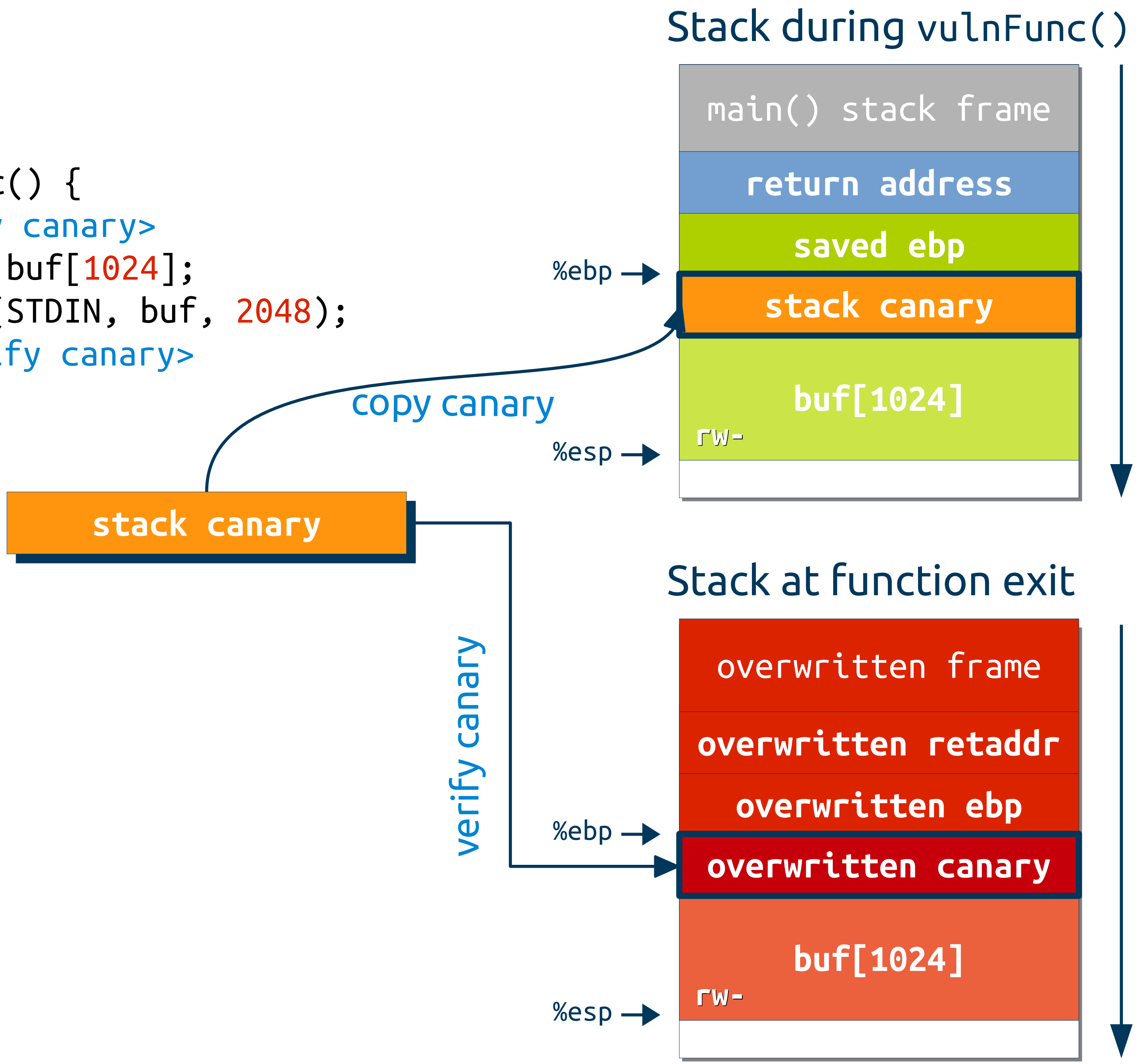
```
void vulnFunc() {
        <copy canary>
        char buf[1024];
        read(STDIN, buf, 2048);
        <verify canary>
}
```



Stack canary / cookie

COMPILER SUPPORT

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



Stack canary / cookie

- Detects linear buffer overflows on stack
  - At function exit
- Corruption of local stack not detected
  - Only if canary / cookie value is overwritten
- Incurs runtime overhead
- Effectiveness relies on secret
  - Leaking, predicting, guessing or brute-forcing might work in special cases

Control Flow Integrity

Attacker model

- DEP & ASLR based on memory model
  - Prevent/complicate attacker access to memory
- Programs execute instructions
  - More involved than use of memory
- Goal: protect program ***execution***

Attacker model

- Let's assume a very **powerful attacker**
  - Can **arbitrarily corrupt data** and pointers
  - Can **read entire address space** of a process
  - Only restriction on attacker
    - No data execution and no code corruption (NX/DEP/W^X)

Attacker model

Can we still prevent arbitrary code execution and code-reuse attacks?

Observations

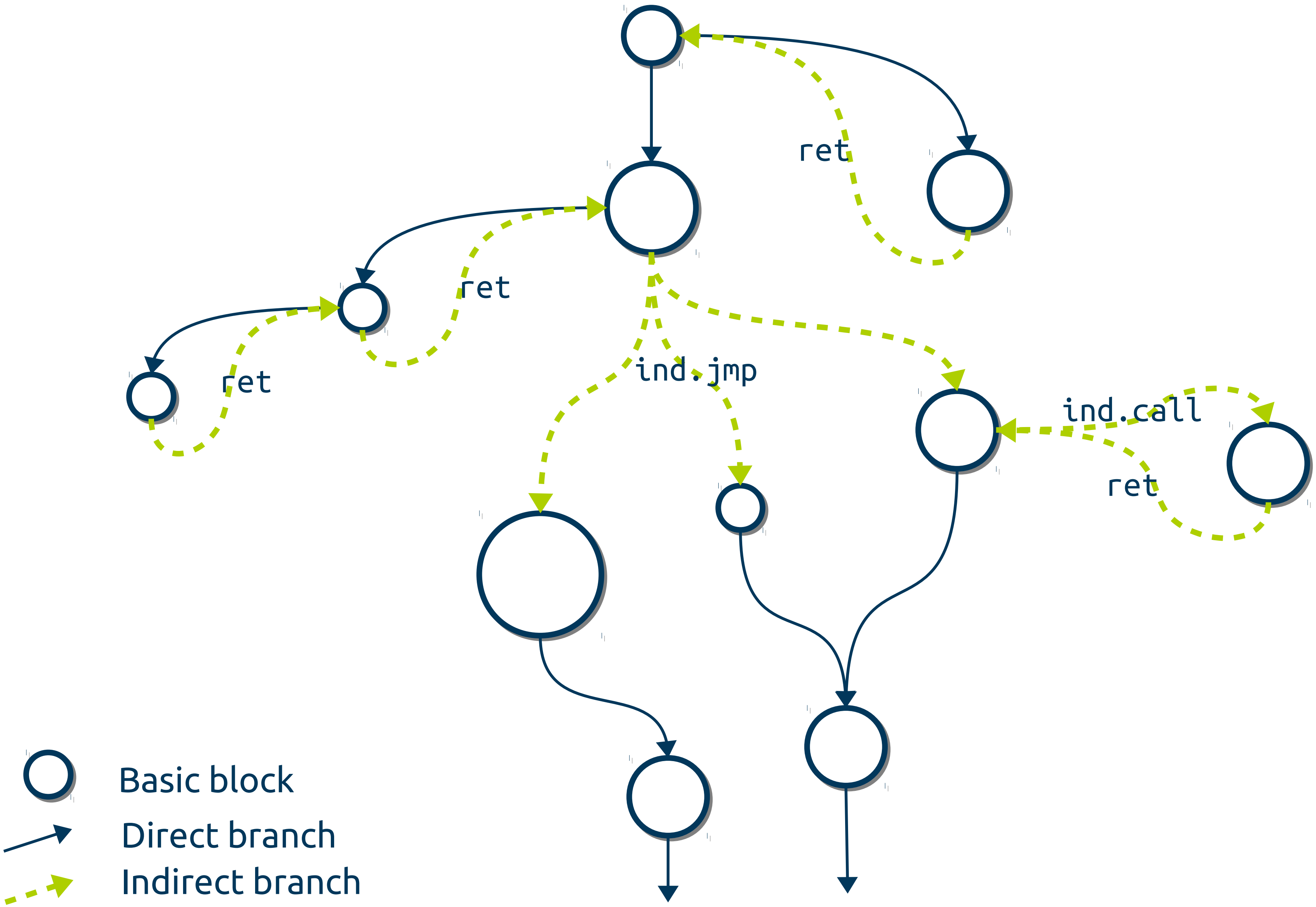
- Attacker needs to hijack control-flow
  - To injected or existing code

Ensure that **control-flow stays** within the intended **legitimate path**

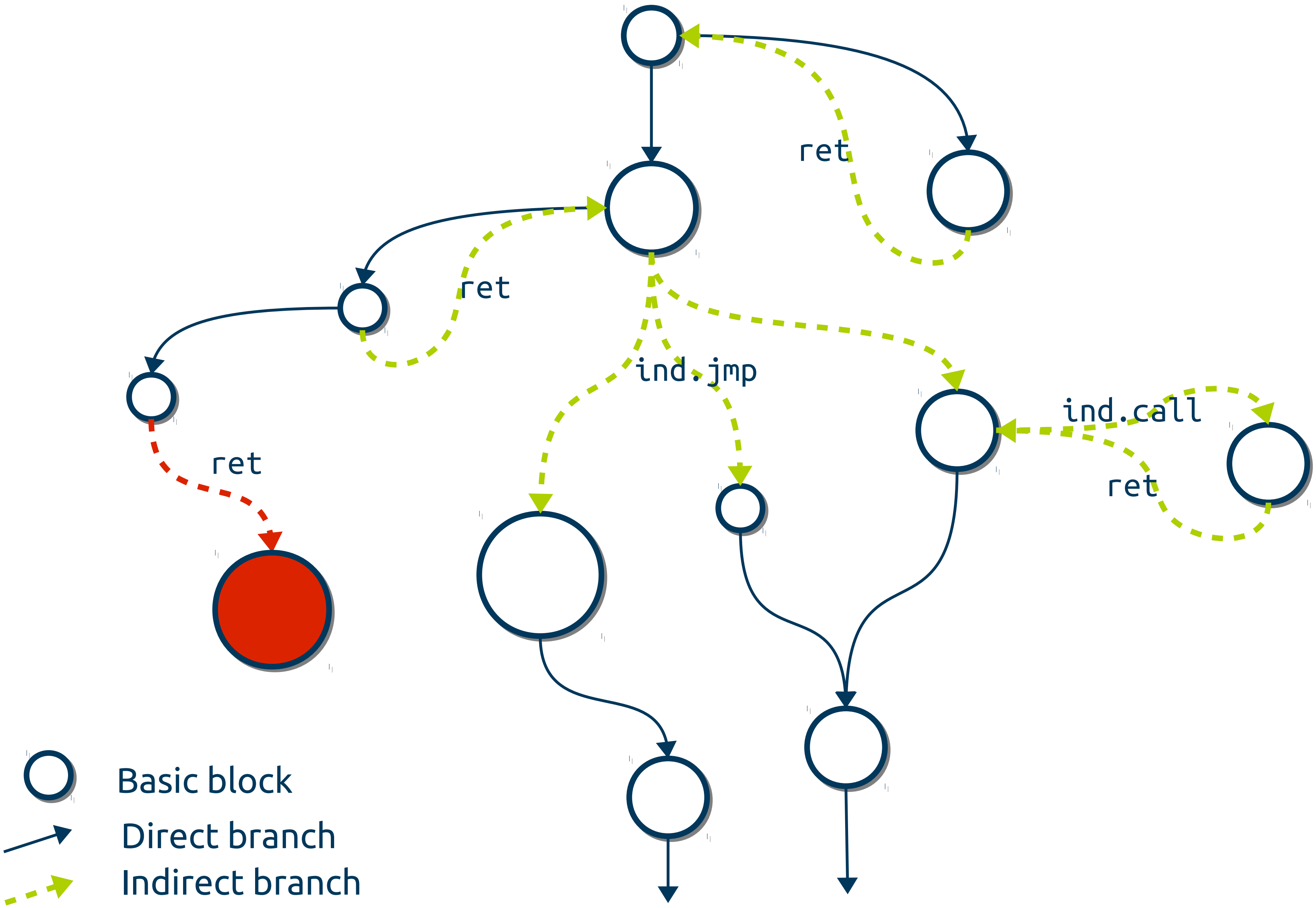
Control-Flow Integrity (CFI)

- Construct a Control-Flow Graph (CFG)
  - Should be as strict as possible
- Ensure that control-flow stays within CFG
- If no path within the CFG can be misused by an attacker then the CFI policy can be considered as secure

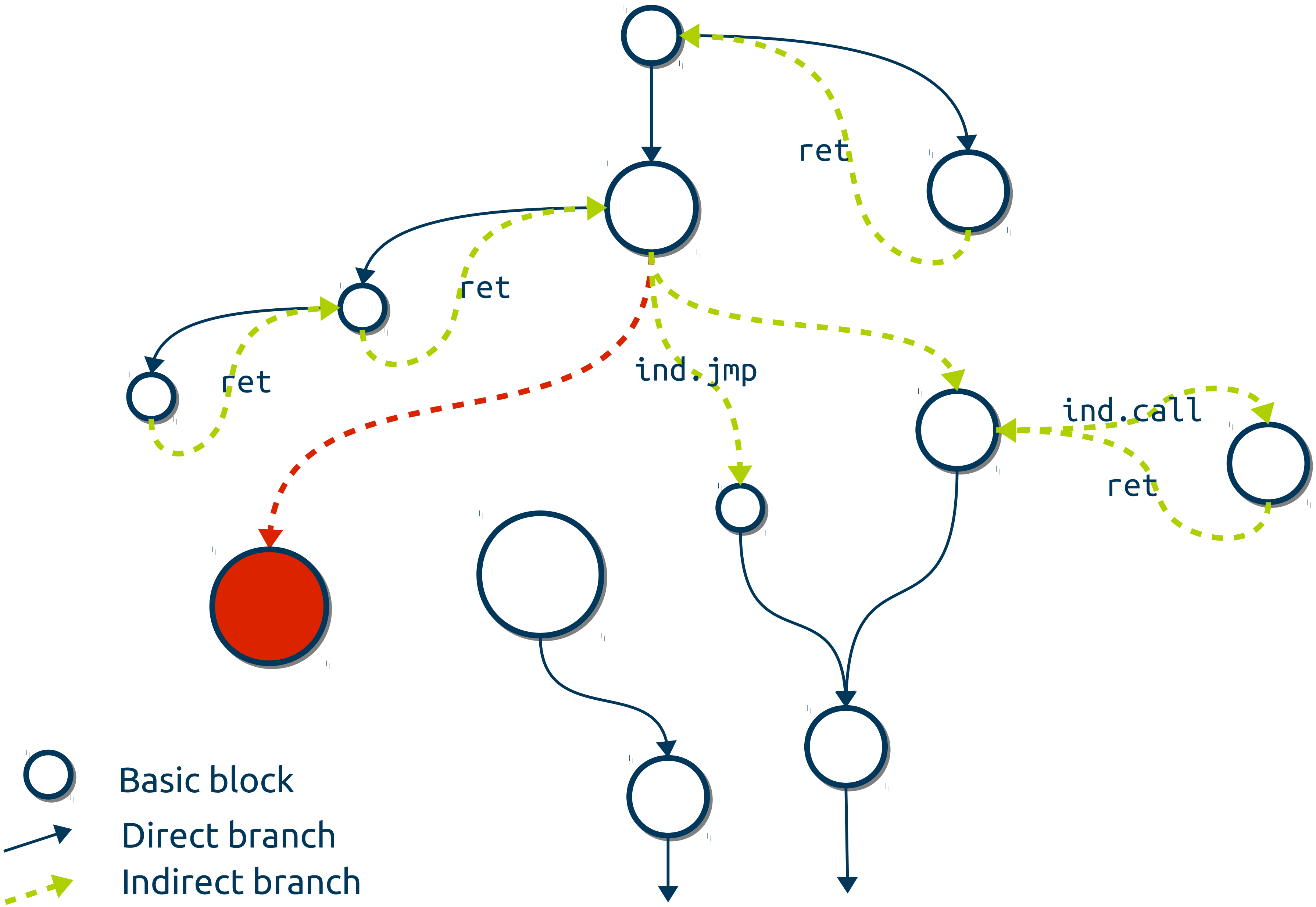
Control-Flow Integrity (CFI)



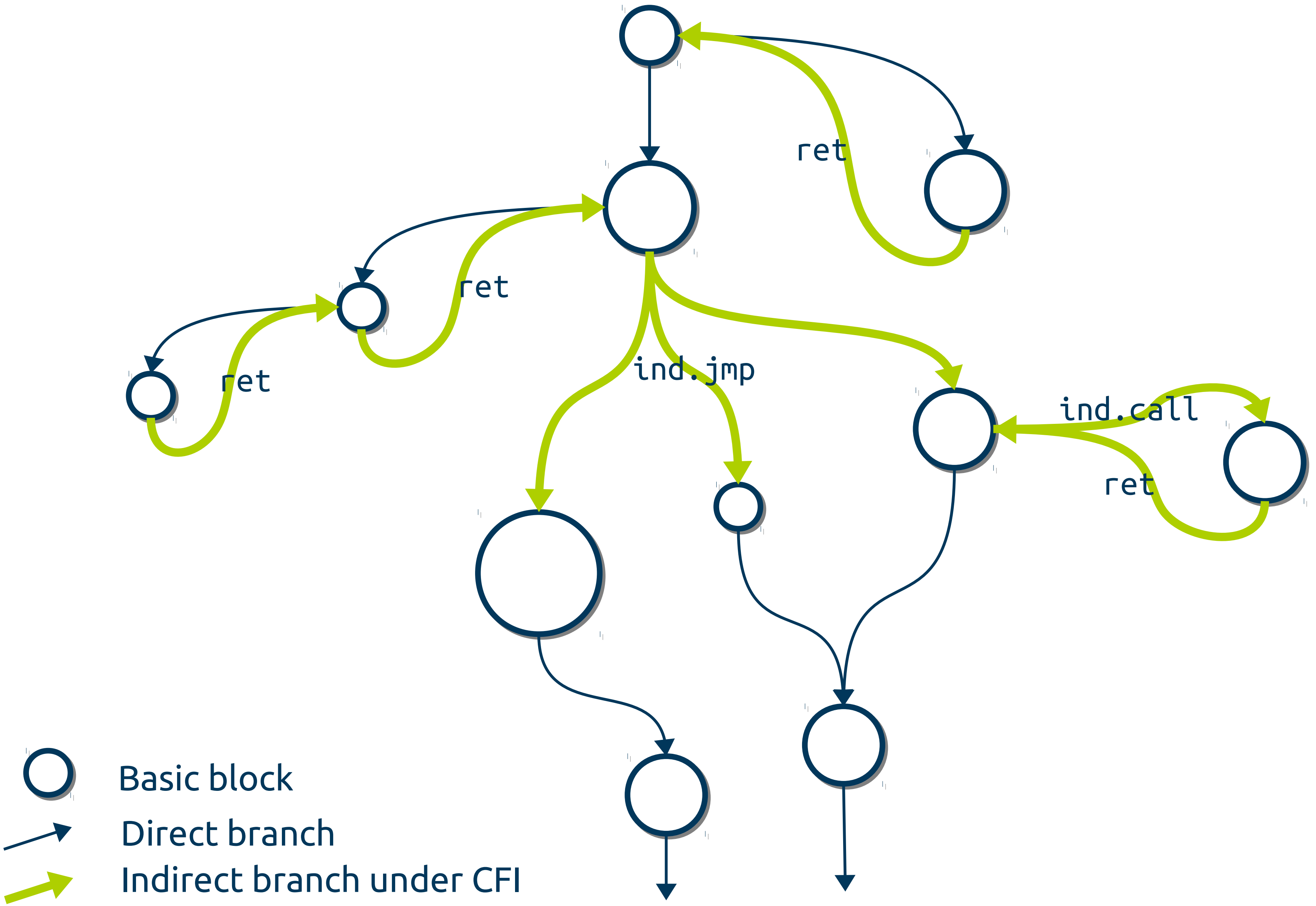
Hijacked control-flow



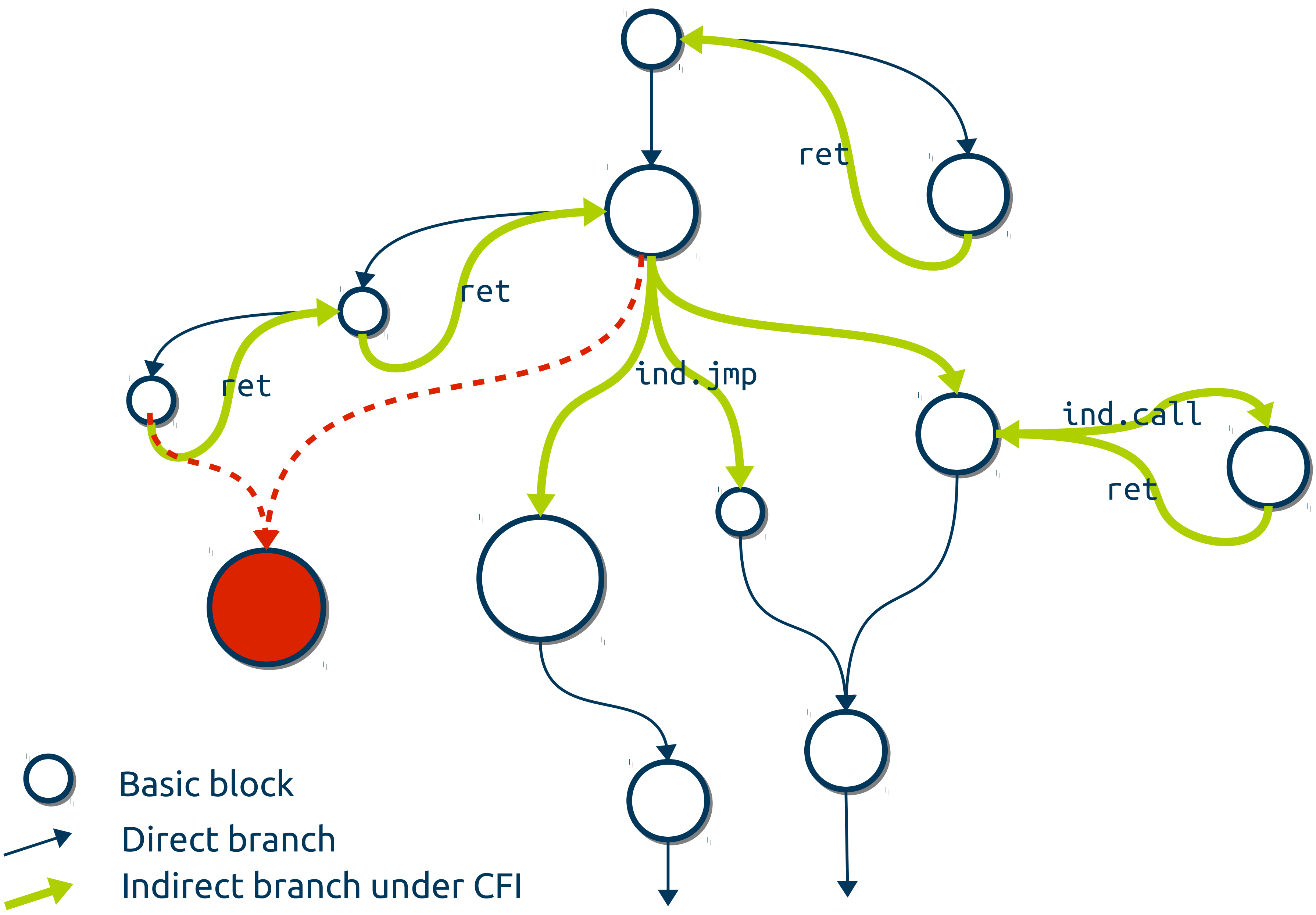
Control-Flow Integrity (CFI)



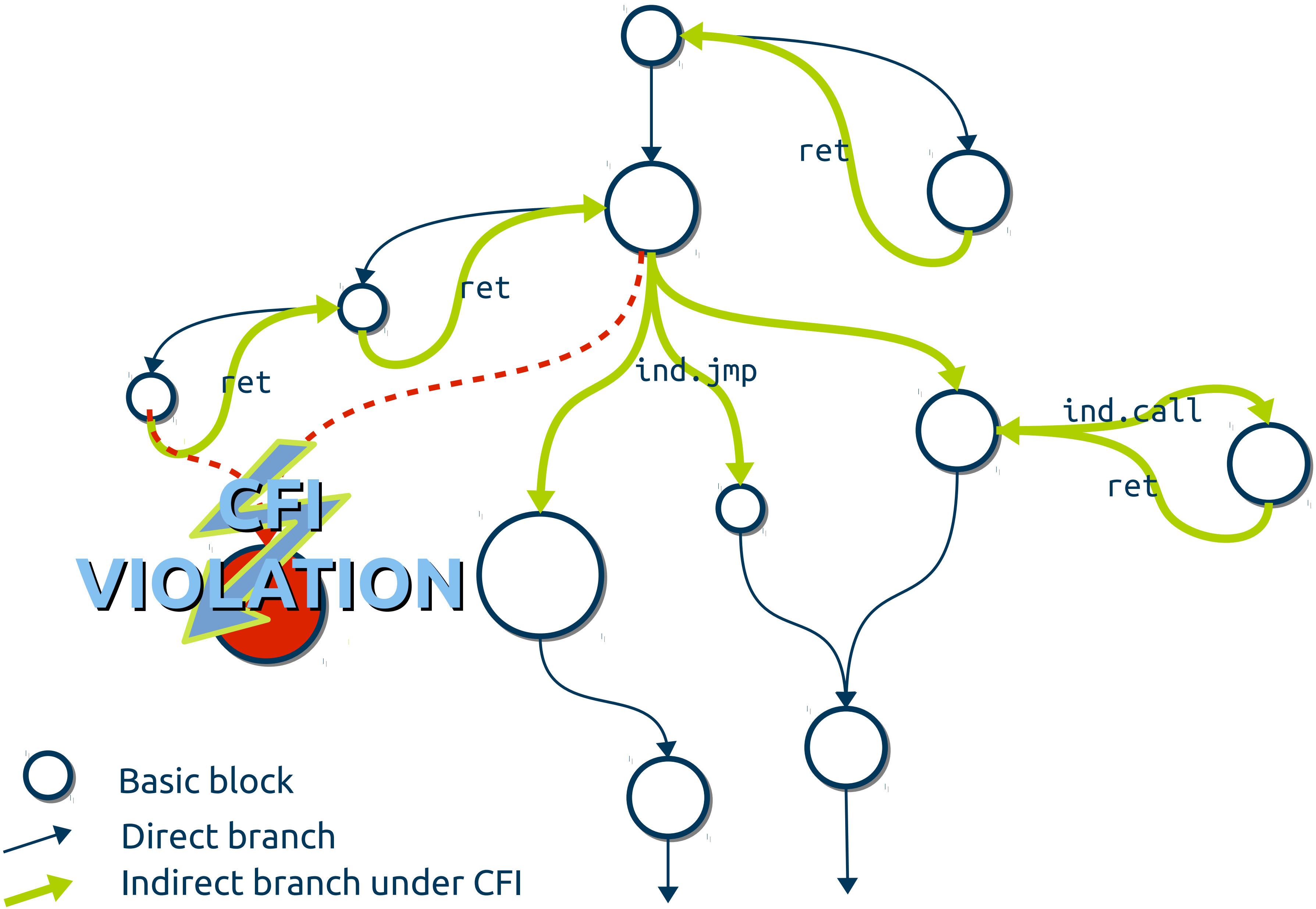
Control-Flow Integrity (CFI)



Control-Flow Integrity (CFI)



Control-Flow Integrity (CFI)



○ Basic block

→ Direct branch

→ Indirect branch under CFI

Control-Flow Integrity (CFI)

- Original publication in 2005
  - “Control-Flow Integrity – Principles, Implementations, and Applications“
    - Abadi, Budiu, Erlingsson, Ligatti, CCS'05
- Many CFI implementations were proposed during recent years
  - Compiler-based, binary-only (static rewriting)

Control-Flow Integrity (CFI)

- Drawbacks of proposed solutions
  - Too permissive CFG due to over-approximation
  - Need to recompile
  - No support for shared libraries

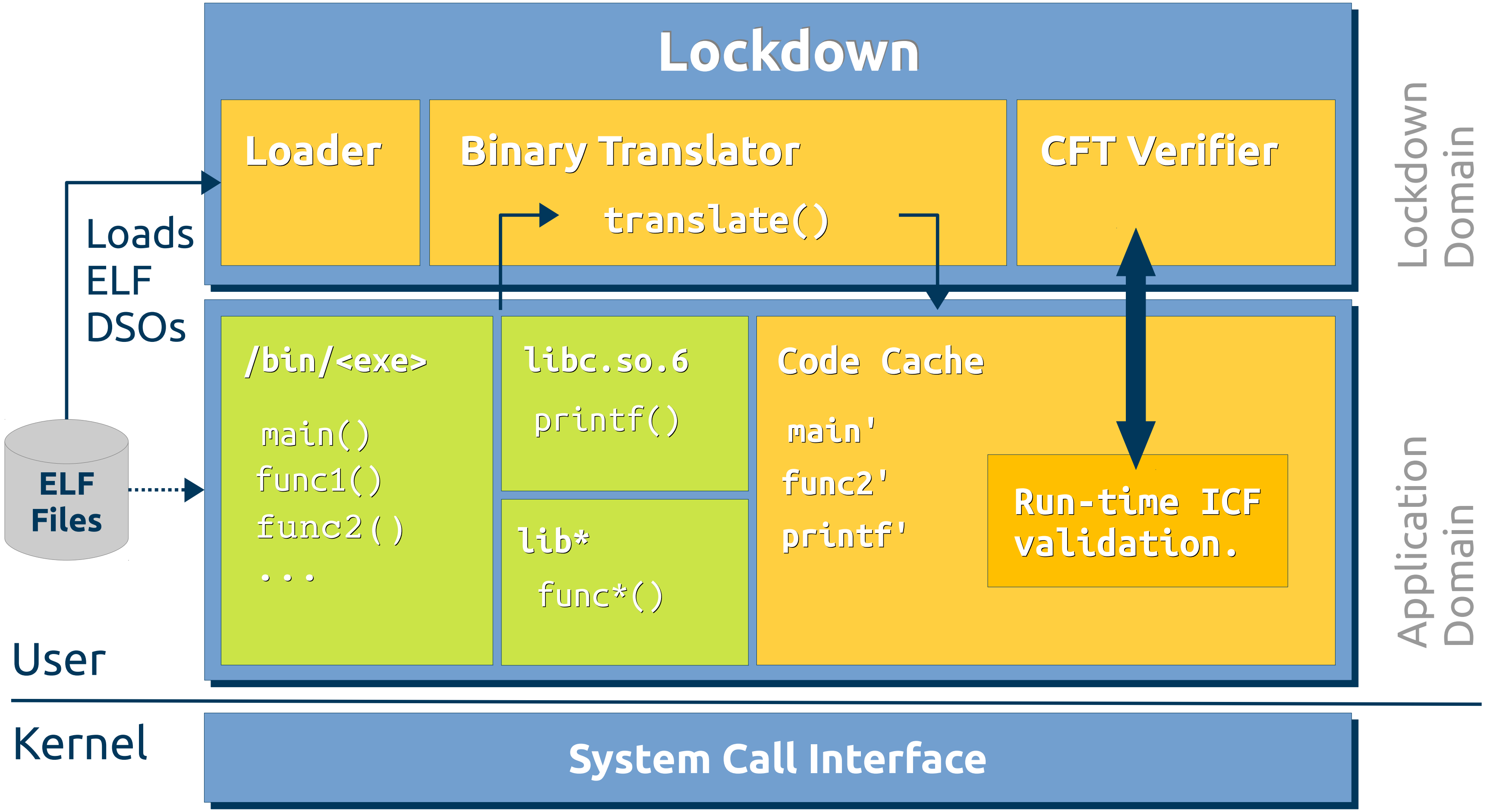
- Most solutions shown to be ineffective
  - “Hardened“ exploits still worked under CFI

Dynamic Control-Flow Integrity (DCFI)

Lockdown – dynamic CFI

- Enforces a strict CFI policy for binaries
- Supports shared libraries & dynamic loading
- Constructs and enforces CFG at runtime
    - Using static and dynamic Information

Lockdown – dynamic CFI



🟩 read only   🟨 readable + executable

CFT: Control-Flow Transfer, ICF: Indirect Control-Flow,
ELF: Executable and Linkable Format, DSO: Dynamic Shared Object

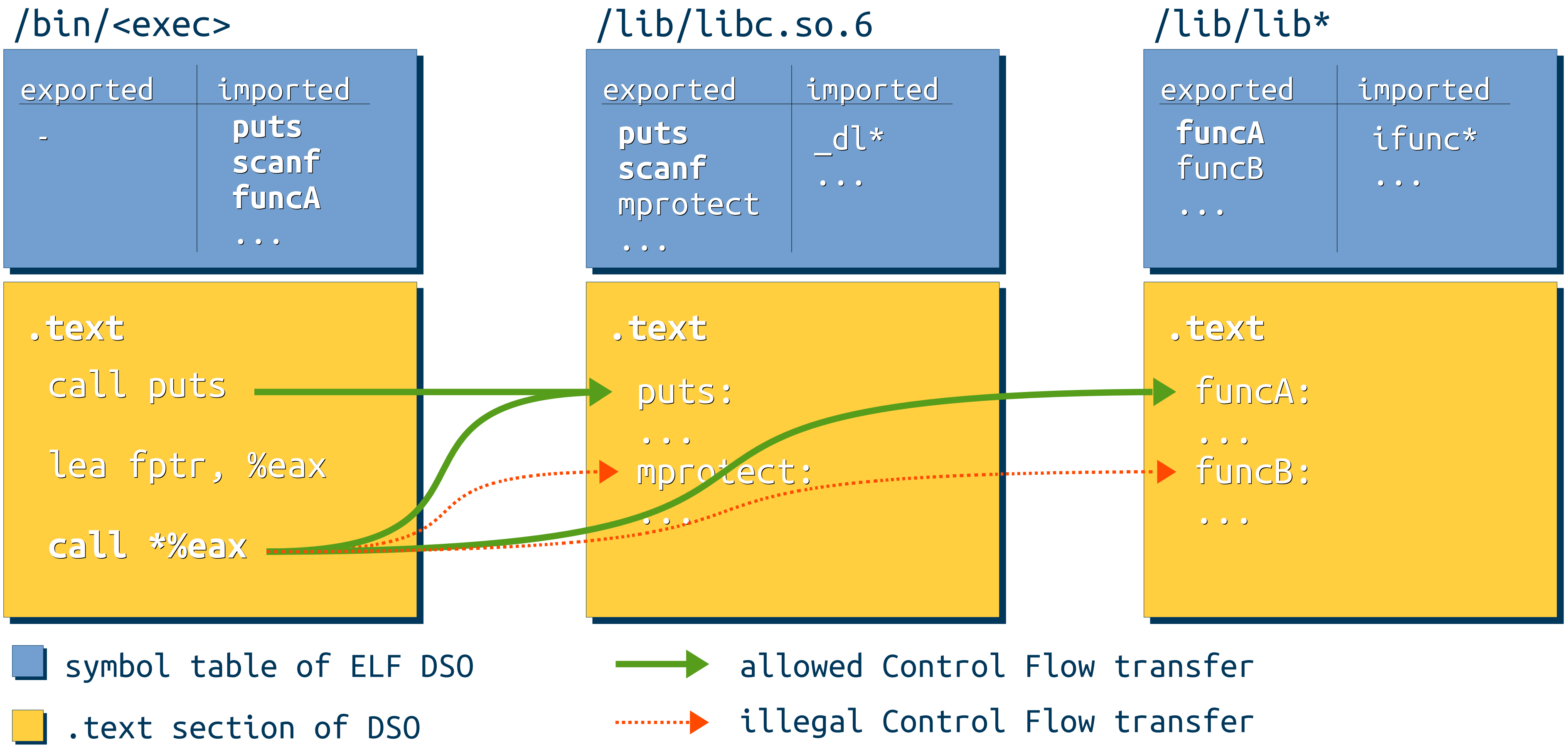
Lockdown – design

- Uses dynamic binary translation to instrument code with additional CFT checks
  - Basically a user-space VM
  - Ensures no untranslated code is ever executed

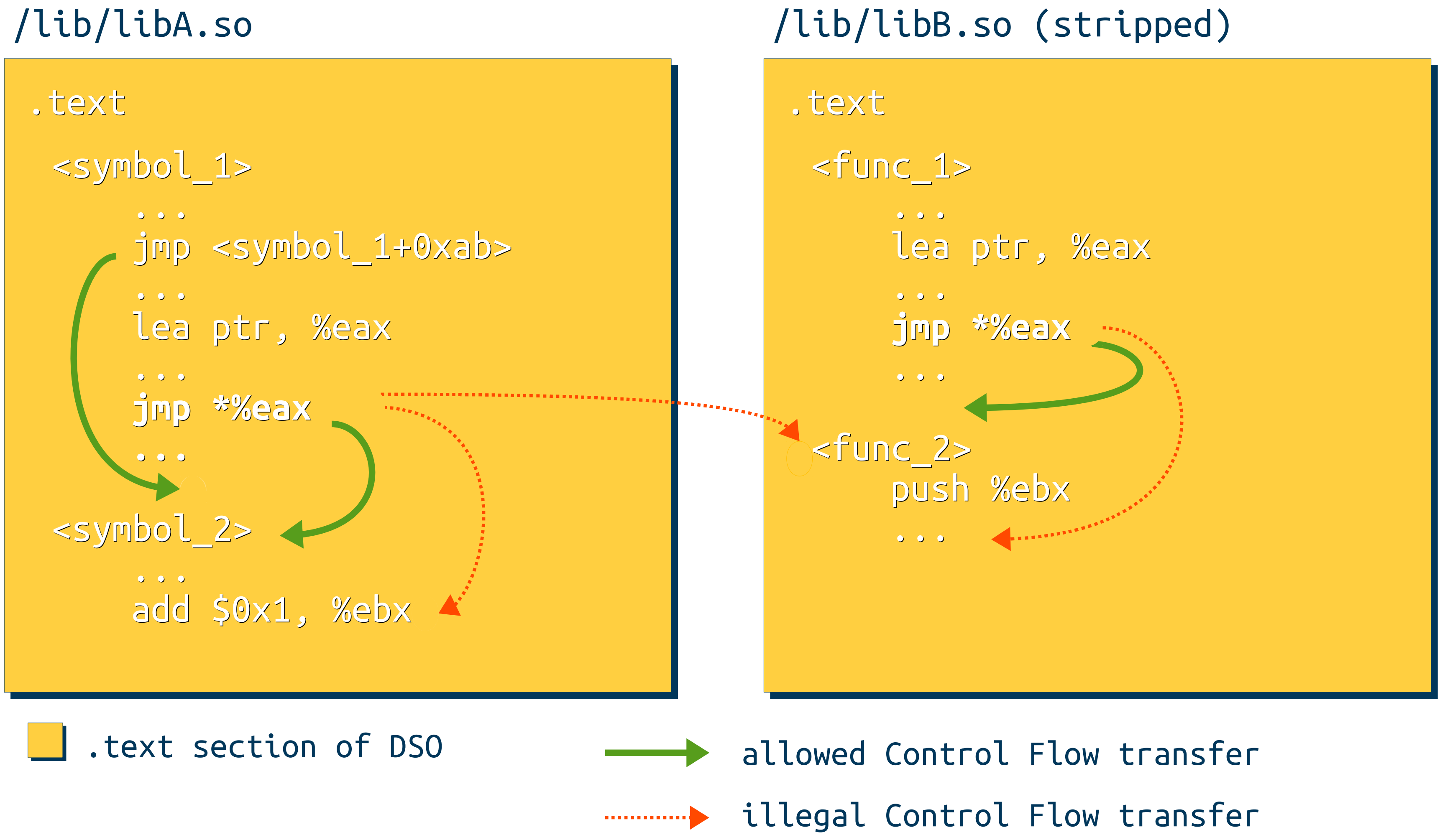
- A trusted loader loads ELF Dynamic Shared Objects (DSOs) and provides symbol information for CFG construction

Mathias Payer, Tobias Hartmann, Thomas R. Gross: “Safe Loading - A Foundation for Secure Execution of Untrusted Programs“ IEEE Symposium on Security and Privacy 2012: 18-32

Lockdown – CFI policy for calls



Lockdown – CFI policy for jumps



/lib/libA.so

```
.text
 <symbol_1>
     ...
     jmp <symbol_1+0xab>
     ...
     lea ptr, %eax
     ...
     jmp *%eax
     ...
 <symbol_2>
     ...
     add $0x1, %ebx
```

/lib/libB.so (stripped)

```
.text
 <func_1>
     ...
     lea ptr, %eax
     ...
     jmp *%eax
     ...
 <func_2>
     push %ebx
     ...
```

🟨 .text section of DSO

→ allowed Control Flow transfer

⇢ illegal Control Flow transfer

Lockdown – CFI policy for returns

- Instrument calls and returns
  - Return address pushed to a shadow stack
  - At return the return address is compared to the value on the shadow stack
    - Resynchronization possible
  - If values don't match raise exception

Lockdown – challenges

- Detection of callbacks & function pointers
    - No information regarding types at runtime
    - If stripped, no extended symbol information
        - Coarser-grained CFG
- Control-flow transfers do not always adhere to the rules presented
- Overhead of CFT checks

Lockdown – performance evaluation

Benchmark	BT overhead	Lockdown overhead
400.perlbench	108.85%	148.16%
401.bzip2	6.65%	6.79%
403.gcc	41.67%	52.22%
433.milc	4.05%	7.92%
444.namd	1.73%	2.08%
Average SPEC CPU2006	**14.64%**	**19.09%**

excerpt

Intel Core i7 CPU 920@2.67GHz with 12GiB
On Ubuntu Linux 12.04.4 LTS 32-bit x86 / gcc 4.6.3

- Total 27 benchmarks (2 benchmarks missing)
- Most benchmarks have an overhead below 20%
- Only 5 benchmarks over 45%

Lockdown - security evaluation

- Nginx use-case
  - **Lockdown effectively prevents exploitation**
  - Attackers have almost no gadgets to use in a code-reuse attack

- Strict static CFI policy still offers plenty of gadgets
  - We were able to “harden“ the nginx exploit to work even under the strict static CFI policy

Conclusion

LAN 127.0.0.1
DNS 192.168.1.1
Upload
Used 0.0
Total 100.0 G
Download
Used 10.0
Total 200.0 G
Connectivity
Security
Security System
CPU 1 10%
RAM
Total 4.0 GB
SWAP
Total 8.0 GB
CPU 2 6%
RAM
Used 2.2 GB
SWAP
Used 1.4 GB
Memory

Conclusion

- Memory errors are still an issue
- Compile-time software hardening is and remains important
- Practical and efficient software hardening techniques are still a hot topic

Thanks!

antonio.barresi@inf.ethz.ch