

CSE509

Computer System Security



2023-02-23

Exploit Mitigations

Michalis Polychronakis

Stony Brook University

Defending against Vulnerability Exploitation

Finding and fixing bugs (*previous lecture*)

Sanitizers, fuzzing, symbolic execution, bug bounties, ...

Who will find the next 0-day?

~~Retrofit memory safety to C/C++~~ → *rewrite critical components in Rust/Go*

Eradicate the root cause of the problem: *memory errors*

Performance and compatibility challenges

No protection against transient execution attacks (!)

Exploit mitigations (*this lecture*)

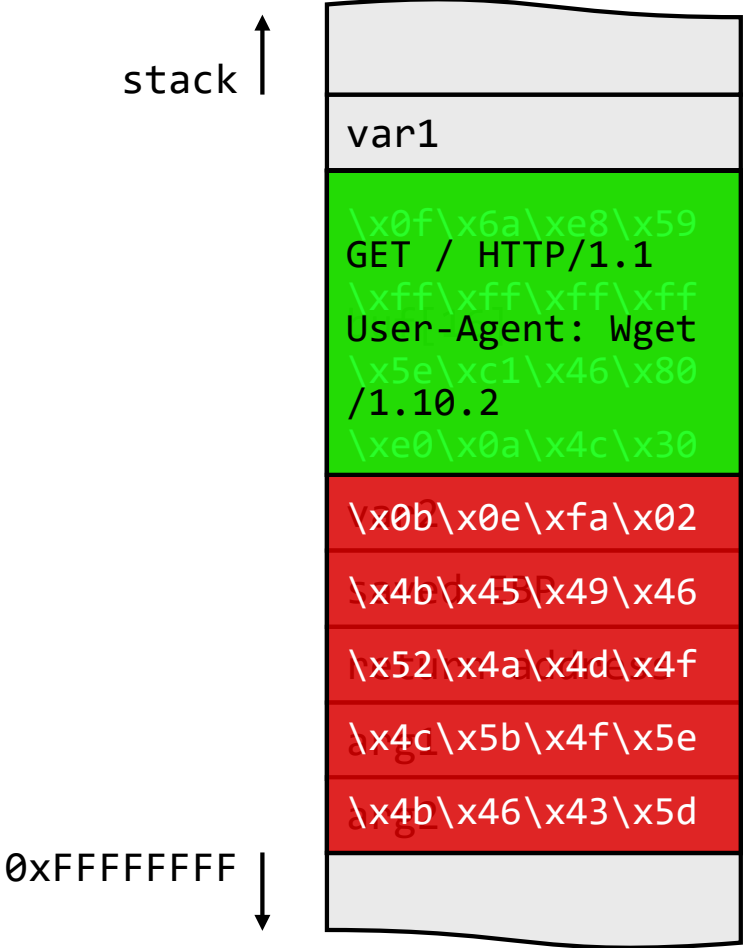
Assuming a vulnerability exists, “raise the bar” for exploitation

DEP, GS, SafeSEH, SEHOP, ASLR, CFI, sandboxing, attack surface reduction, ...

Exploit Mitigations vs. Sanitizers

	Exploit Mitigations	Sanitizers
The goal is to ...	Mitigate attacks	Find vulnerabilities
Used in ...	Production	Pre-release
Performance budget ...	Very limited	Much higher
Policy violations lead to ...	Program termination	Problem diagnosis
Violations triggered at location of bug ...	Sometimes	Always
Tolerance for FPs is ...	Zero	Somewhat higher
Surviving benign errors is ...	Desired	Not desired

(Very Simple) Buffer Overflow Exploitation



← Code injection

Shellcode

- spawn shell
- listen for connections
- add user account
- download and execute malware**

Stack Canaries

Goal: prevent control flow hijacking via return address overwrite

aka:

Stack Cookie

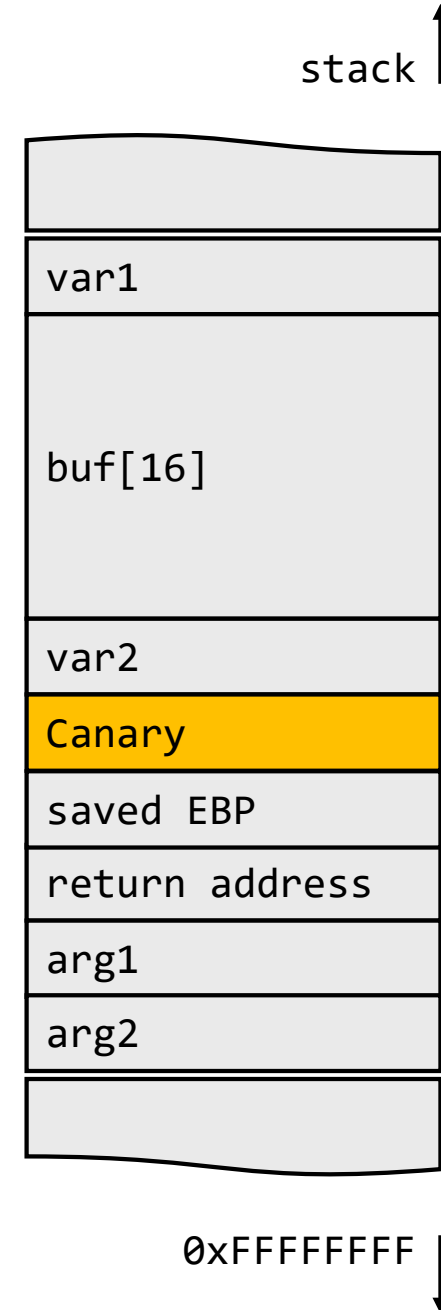
Stack Smashing Protector

Microsoft's /GS

Add randomly generated (secret) value before return address

Function epilogue code checks the canary value before returning

Exception if it has an unexpected value



buf[]	Canary	Saved EIP
-------	--------	-----------

Before overflow

buf[]	0xDEADBEEF	0xBFFFFFFC2A
-------	------------	--------------

After overflow

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA	0x41414141	0x42424242
----------------------------------	------------	------------

0x41414141 != 0xDEADBEEF → Terminate process

StackGuard (1998)

First implementation of stack canaries (GCC)

Random canary

Taken from /dev/urandom when the program starts

Terminator canary

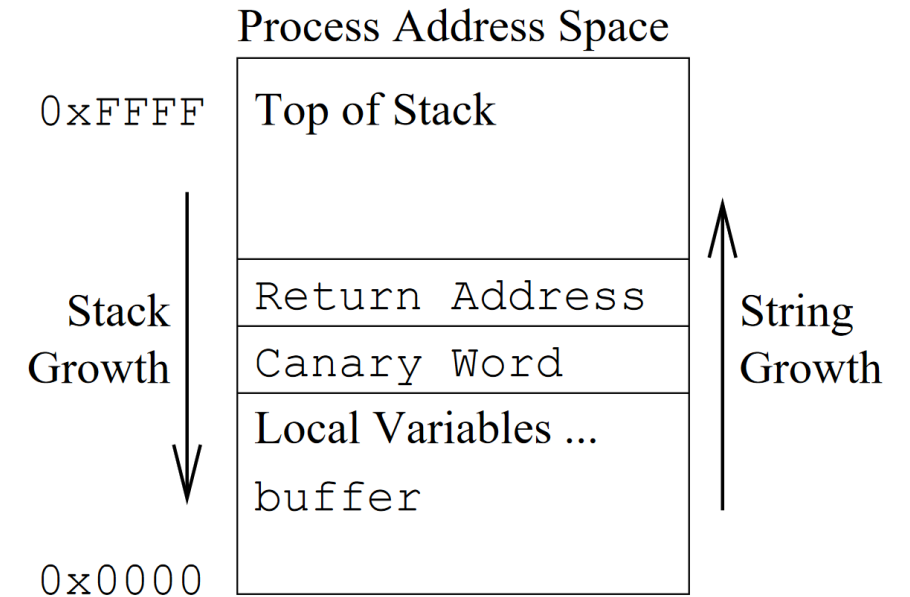
Contains NULL (0x00), CR (0x0d), LF (0x0a), and EOF (0xff) → should terminate most string operations

Random XOR canary

Emsi's Vulnerability: overwrite the return address with arbitrary write (write-what-where instead of linear overflow)

Solution: XOR the canary with the saved return address

Attacker cannot modify the return address without invalidating the canary



ProPolice (IBM Research Japan, 2000)

Security improvement over StackGuard: local variable sorting

Move all buffers (arrays) above local variables → prevent local variable overwrite

Eventually merged into GCC and became the Stack Smashing Protector (SSP) feature (`-fstack_protector`)

`-fstack-protector` stack protection for functions that contain an array larger than 8 bytes

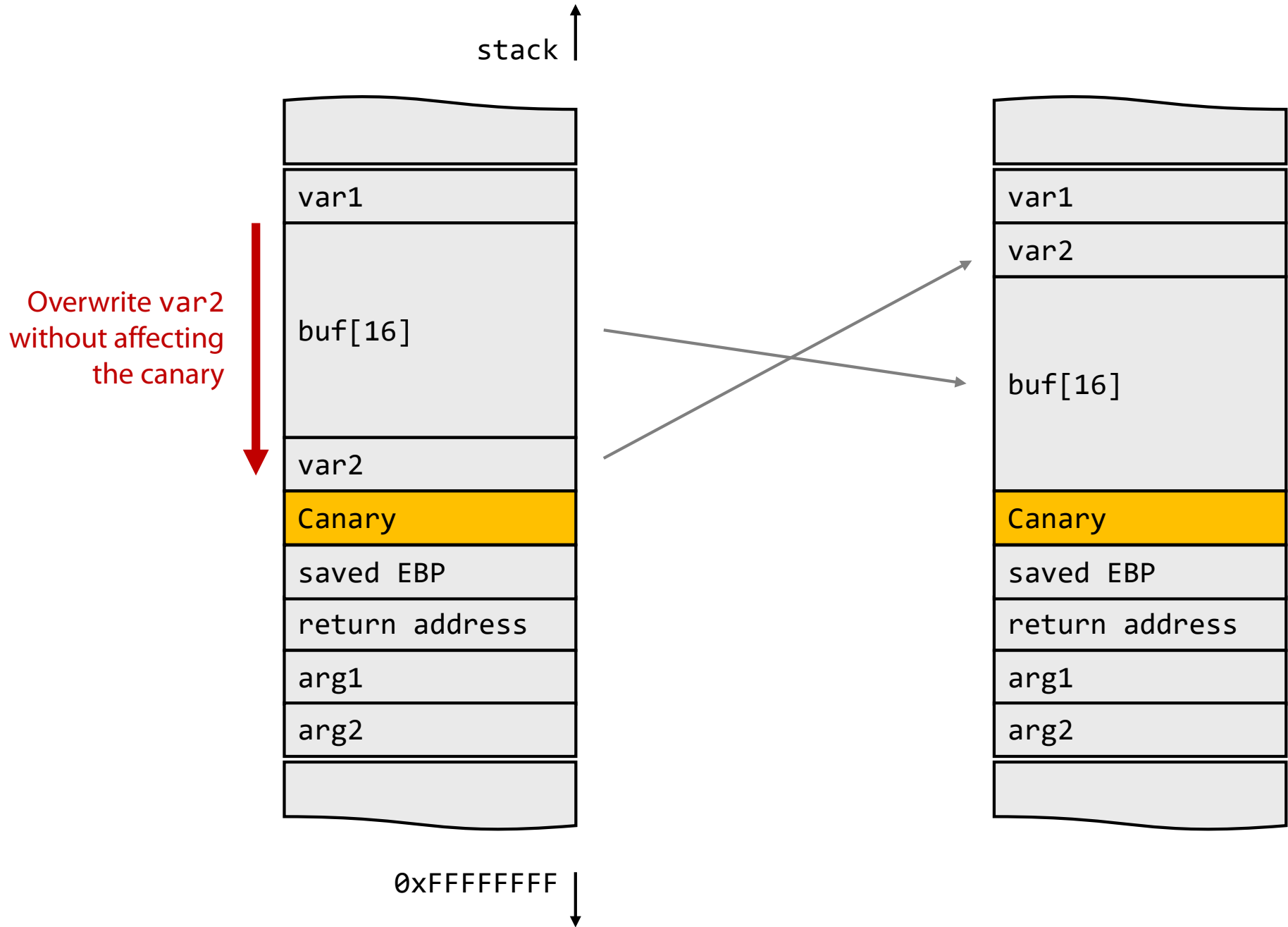
`-fstack-protector-all` stack protection for all functions

`-fstack-protector-strong` introduced by Google in 2012 → strike a (performance) balance between `-fstack_protector` and `-fstack_protector-all`

Microsoft's /GS

Introduced in Visual Studio 2002, deployed in Windows XP SP2

Although 5 years after StackGuard, it beat the Linux/FOSS community into mainstream adoption by several years



Canary Value Brute Forcing

The canary is generated dynamically at the creation of each thread

Typically stored in the Thread Local Storage (TLS) area

Whenever a new frame is created, the canary is pushed from the TLS into the stack

Whenever a process is forked, it inherits the address space of its parent

All in-memory code and data, including canaries placed in stack frames and the TLS

Problematic for forking servers: attackers can brute-force the canary

- 1) Need to be able to force child processes to be forked from the same parent process
- 2) Need to be able to check if any of these child processes has crashed or not

Solution: randomize canary values right after fork

Update the canaries in both the TLS and all inherited stack frames in the child process



...



→ Crash

...



→ No Crash

256⁴ → 256*4

...



→ Crash

...



→ No Crash

...

$4 * 256 = 2^{10} = 1024$ possibilities → just 512 tries on average (!)

Instead of $256^4 = 2^{32} = 4$ billion possibilities

$8 * 256 = 2048$ in 64-bit architectures

Nginx 1.3.9 < 1.4.0 - Chunked Encoding Stack Buffer Overflow

```
def find_canary
  # First byte of the canary is already known
  canary = "\x00"

  print_status("#{peer} - Assuming byte 0 0x%02x" % 0x00)

  # We are going to bruteforce the next 3 bytes one at a time
  3.times do |c|
    print_status("#{peer} - Bruteforcing byte #{c + 1}")

    0.upto(255) do |i|
      data = random_chunk_size(1024)
      data << Rex::Text.rand_text_alpha(target['CanaryOffset'] - data.size)
      data << canary
      data << i.chr

      unless send_request_fixed(data).nil?
        print_good("#{peer} - Byte #{c + 1} found: 0x%02x" % i)
        canary << i.chr
        break
      end
    end
  end
end
```

Other Stack Smashing Defenses

StackShield

Copy the return address into a safe memory area when a function is called

Restore the return address upon returning from a function

Even if the return address on the stack is altered, it has no effect since the original return address is remembered

Libsafe

Re-implementation of dangerous C functions such as `strcpy()`

Imposes a limit on the copied bytes to prevent return address overwrite

Limit determined based on the distance between the address of the buffer and the corresponding frame pointer

Shared library that is preloaded (`LD_PRELOAD`) to intercept C library function calls

FORTIFY_SOURCE (Red Hat, 2004)

Compilation time + run time buffer overflow detection

```
char buf[5];
```

```
/* 1) Known correct. No runtime checking is needed, memcpy/strcpy are called */
```

```
memcpy (buf, foo, 5);  
strcpy (buf, "abcd");
```

```
/* 2) Not known if correct, but checkable at runtime. The compiler knows the number  
of bytes in object, but doesn't know the length of the actual copy. Alternative  
functions __memcpy_chk or __strcpy_chk are used that check for buffer overflow. */
```

```
memcpy (buf, foo, n);  
strcpy (buf, bar);
```

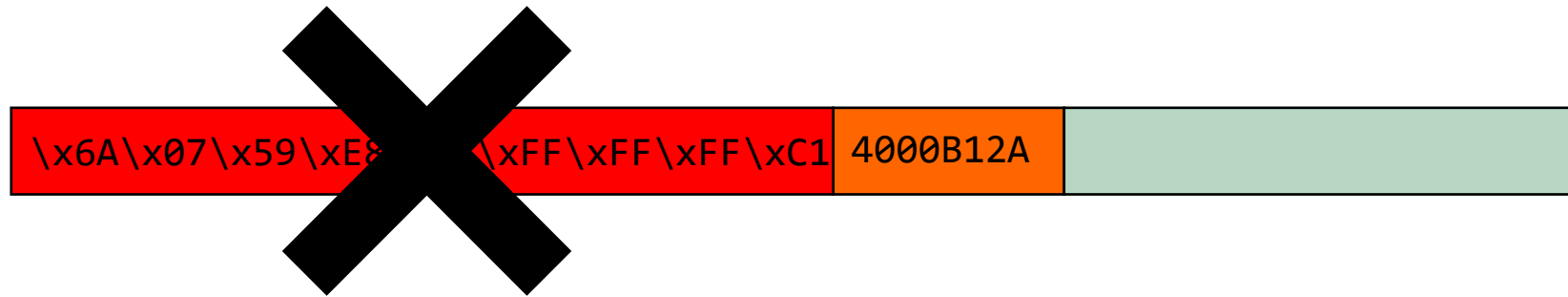
```
/* 3) Known incorrect. The compiler can detect buffer overflows at compile time. */
```

```
memcpy (buf, foo, 6);  
strcpy (buf, "abcde");
```

```
/* 4) Not known if correct, not checkable at runtime. The compiler doesn't know the  
buffer size, no checking is done. Overflows will go undetected in these cases. */
```

```
memcpy (p, q, n);  
strcpy (p, q);
```

Non-Executable Memory (aka: NX, PaX, Exec Shield, W^X, DEP, ...)



Initial implementations were software-only

x86 processors (since 80286) supported memory segmentation, which *can* distinguish between readable and executable memory

Unfortunately almost all x86 OS use a flat memory model → could not use this capability

Hardware x86 support for NX bit introduced by AMD64 (x86-64)

Followed by Intel (Pentium 4 processors based on later iterations of the Prescott core)

For 32-bit, need PAE (Physical Address Extension, 32 to 36 bit)

Non-Executable Stack (Solar Designer, 1997)

x86 Linux kernel patch to prevent code execution in stack pages

Sun Solaris 2.6 (SPARC) also introduced a similar feature

At that time, x86 did not have an execute bit → readable pages were implicitly executable

Other CPUs did have an X bit, but they were far less popular

Solution: use x86 segments to isolate the stack and mark it as NX

Zero performance overhead

No need to re-compile programs

Non-executable stack was the main motivation for ret2libc attacks

PaX PAGEEXEC

Early software-only Linux implementation of W^X

Mark all data pages as non-executable (not just stack pages)

No NX bit in early x86: emulate it

Mark pages either as “non-present” or “supervisor”

User-space memory accesses raise page faults

Page fault handler distinguishes between instruction fetches and data accesses (by comparing the fault address to that of the instruction that raised the fault)

Significant runtime overhead due to the additional page faults

PaX SEGMEXEC: alternative implementation using x86 segmentation

Divide user space into two halves: one for data access and one for execution

ExecShield: similar approach by Red Hat

Data Execution Prevention (DEP)

Introduced in Windows XP SP2

Hardware-only implementation: disabled if the x86 processor did not support NX

Enabled by default for critical Windows services

Applications can opt-in (`SetProcessDEPPolicy()` or `/NXCOMPAT`)

Runtime access to the NX bit is exposed through the Win32 API

`VirtualAlloc()` and `VirtualProtect()` → flag individual pages as executable or non-executable

Precious API functions for ROP exploits (same for `mprotect()` in Linux)

Address Space Layout Randomization

Hinders code reuse attacks by randomizing the location of code and data → unpredictable target addresses (shellcode, ROP gadgets)

Main executable, stack, heap, libraries

DEP is highly dependent on ASLR

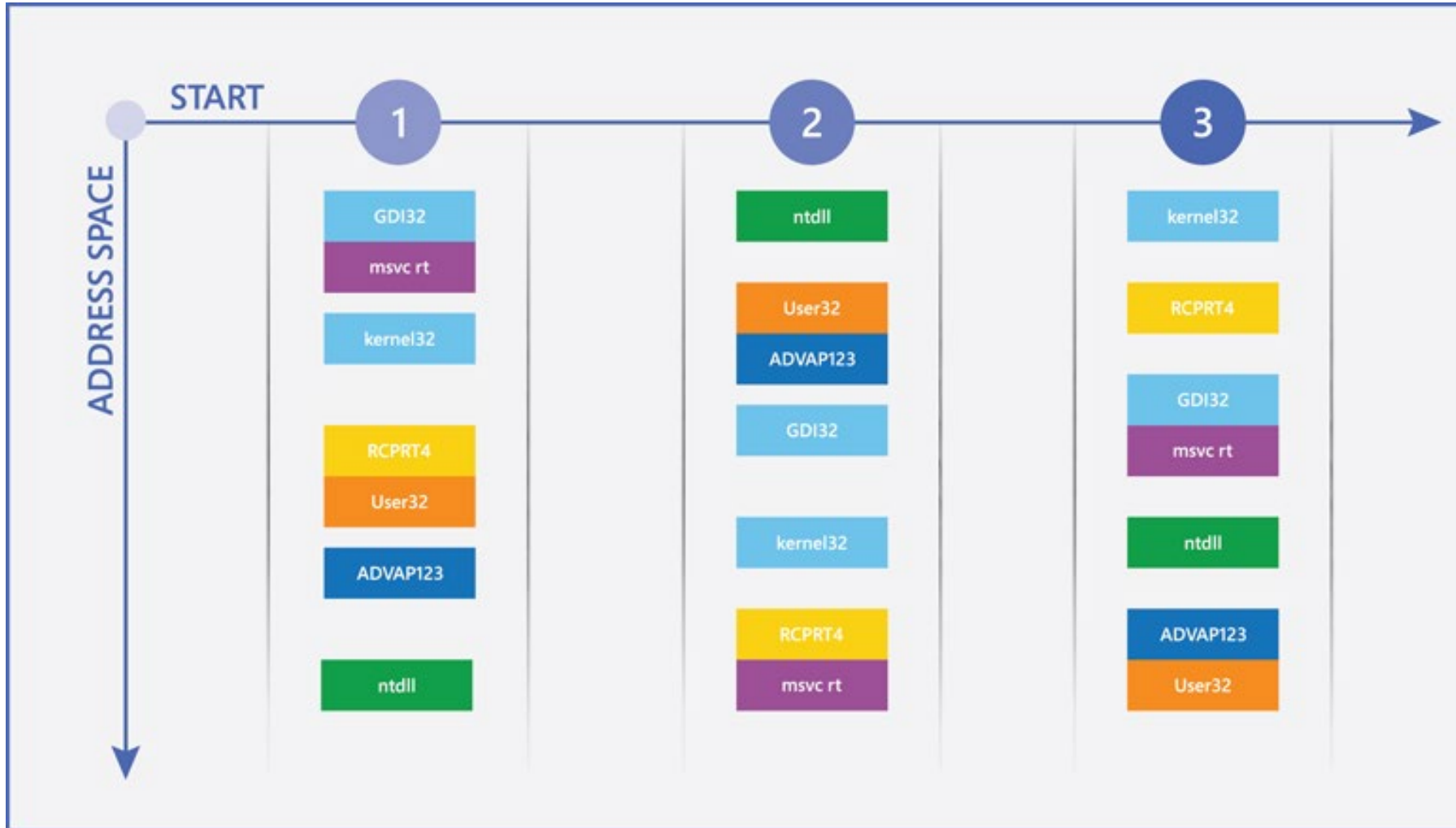
ROP/ret2libc payloads are trivial to construct without ASLR

Initial implementation: PaX ASLR (2001)

OpenBSD version 3.4 in 2003 was the first mainstream OS to support ASLR

Followed by Linux in 2005 (initial version was weaker than PaX/ExecShield)

Introduced in Windows with Vista in 2007



ASLR Limitations

During its early deployment, ASLR was not always fully adopted

- Only 66 out of 1,298 binaries in /usr/bin (2011)

- Only 2 out of 16 third-party Windows applications (2010)

- Non-ASLR'ed binaries can still occasionally be found even today on the latest systems

ASLR-enabled applications sometimes have statically mapped DLLs

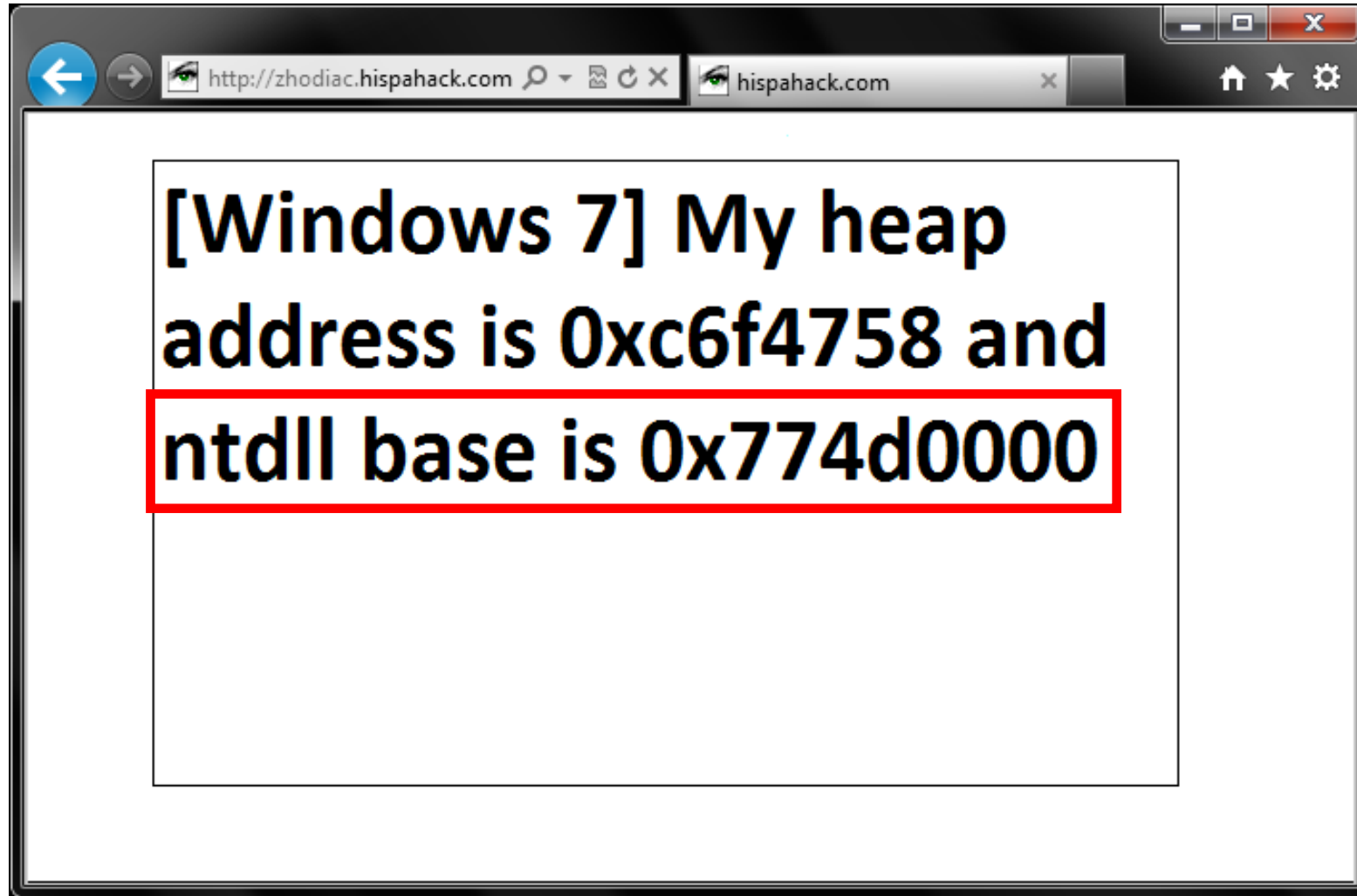
- Microsoft's EMET mandatory randomization: force relocation even if DLL opt-out

More importantly: Information leaks break ASLR

- Dynamically infer a DLL's load address through a memory leak vulnerability

- Adjust offsets of ROP gadgets/ret2libc addresses accordingly

Information Leaks Break ASLR



Typical ROP exploits

First-stage ROP code to bypass DEP

1. Allocate/set W+X memory (`VirtualAlloc`, `VirtualProtect`, ...)
2. Copy embedded shellcode into the newly allocated area
3. Execute!

Pure-ROP exploits are also possible

Example: in-the-wild exploit against Adobe Reader XI (CVE-2013-0640)

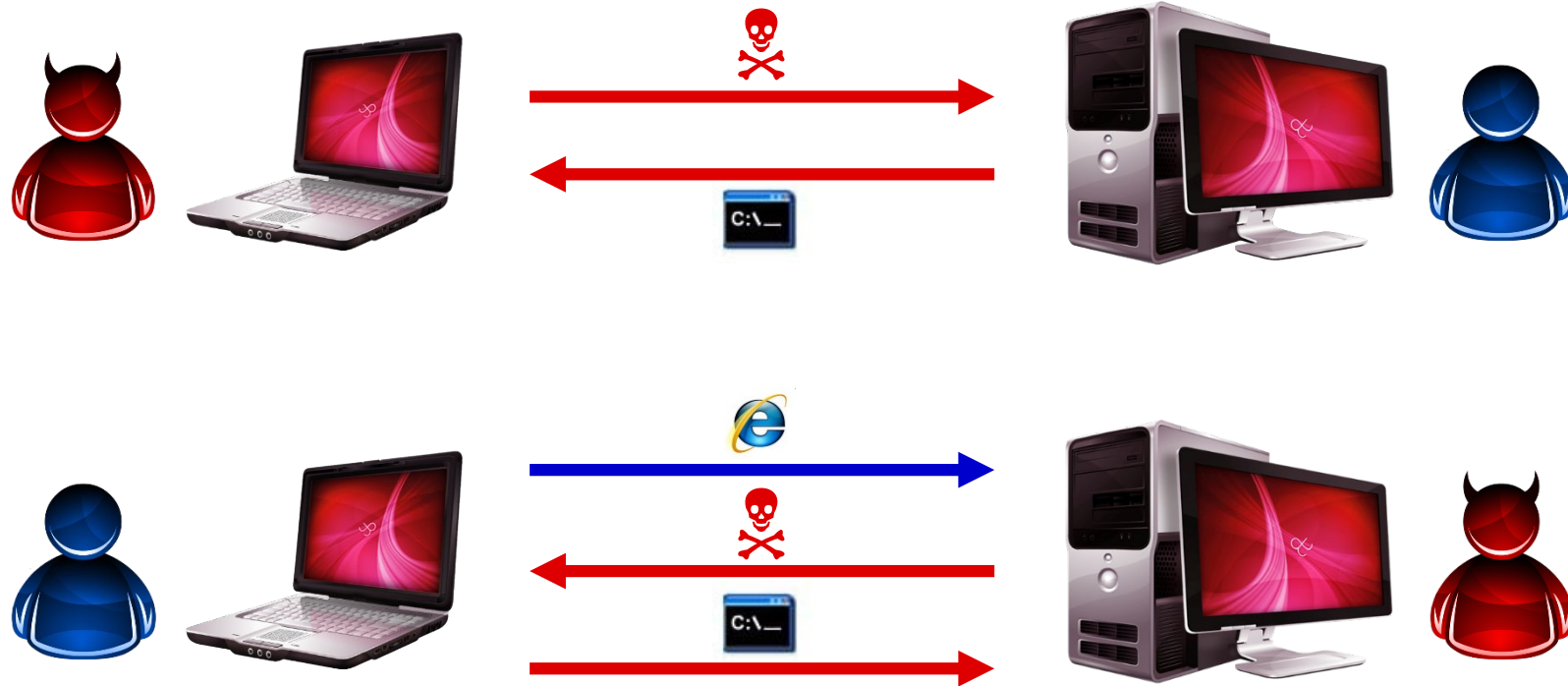
The complexity of ROP exploit code increases

ROP exploit mitigations in Windows 8/8.1

Control Flow Integrity in Windows 10

Dynamic ROP payload construction (JIT-ROP)

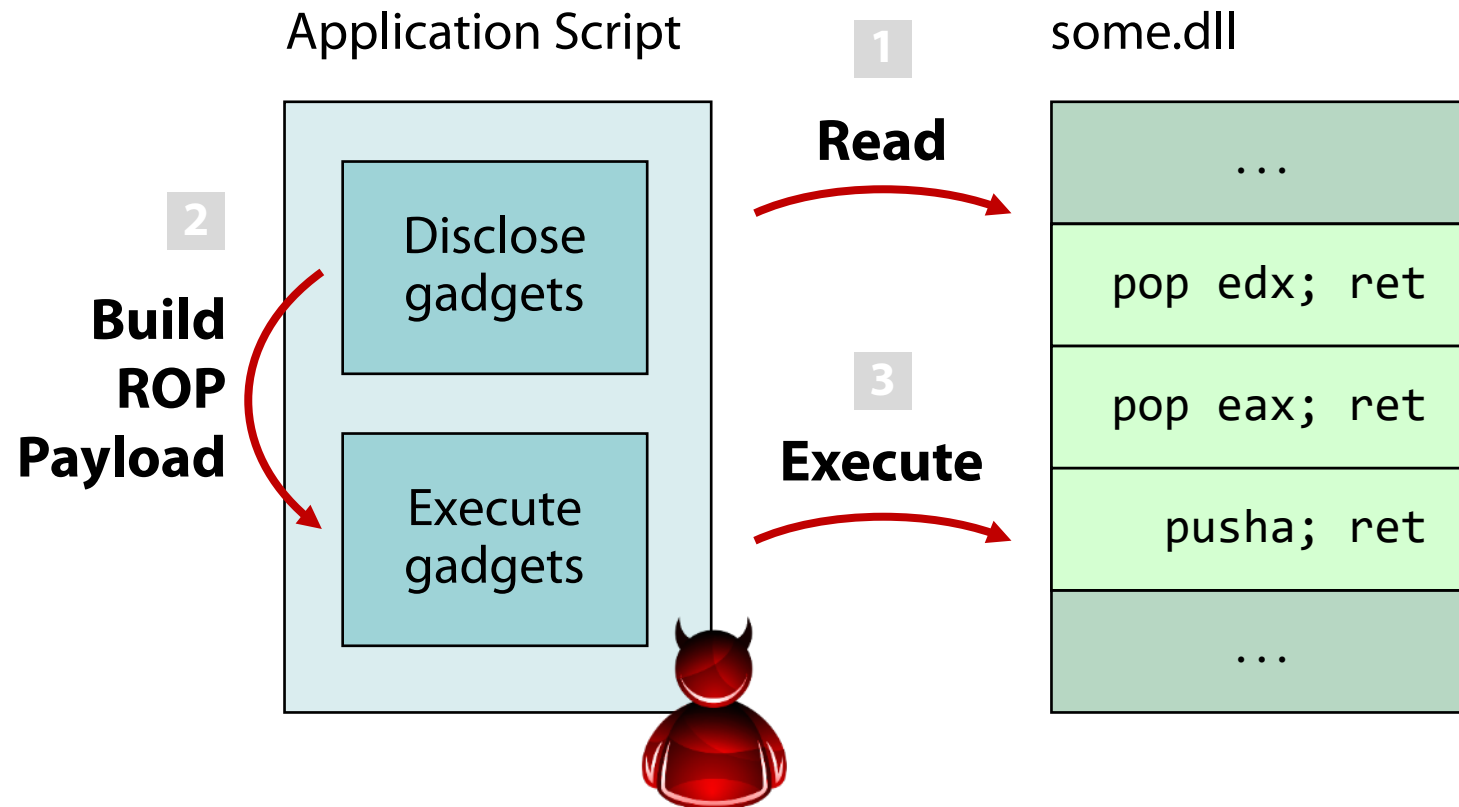
Remote Exploitation: Server-side vs. Client-side



Client-side applications (web browsers, document readers, etc.) offer a more feature-rich environment for attackers to launch exploits through script code and program features

Just-in-time Code Reuse (JIT-ROP)

Build a ROP payload *on-the-fly* using malicious JavaScript code that scans the code segments of the process for ROP gadgets through a memory leak vulnerability



Memory Page Permissions

Old x86 CPUs have 1 bit per page: **W**

A page can be writable or not, but is always executable → *Code injection*: write data into memory and then execute it

Modern CPUs have 2 bits per page: **W, X**

W^X: A page can be marked as writable but *non-executable*
Code injection is prevented, but *code reuse* is still possible

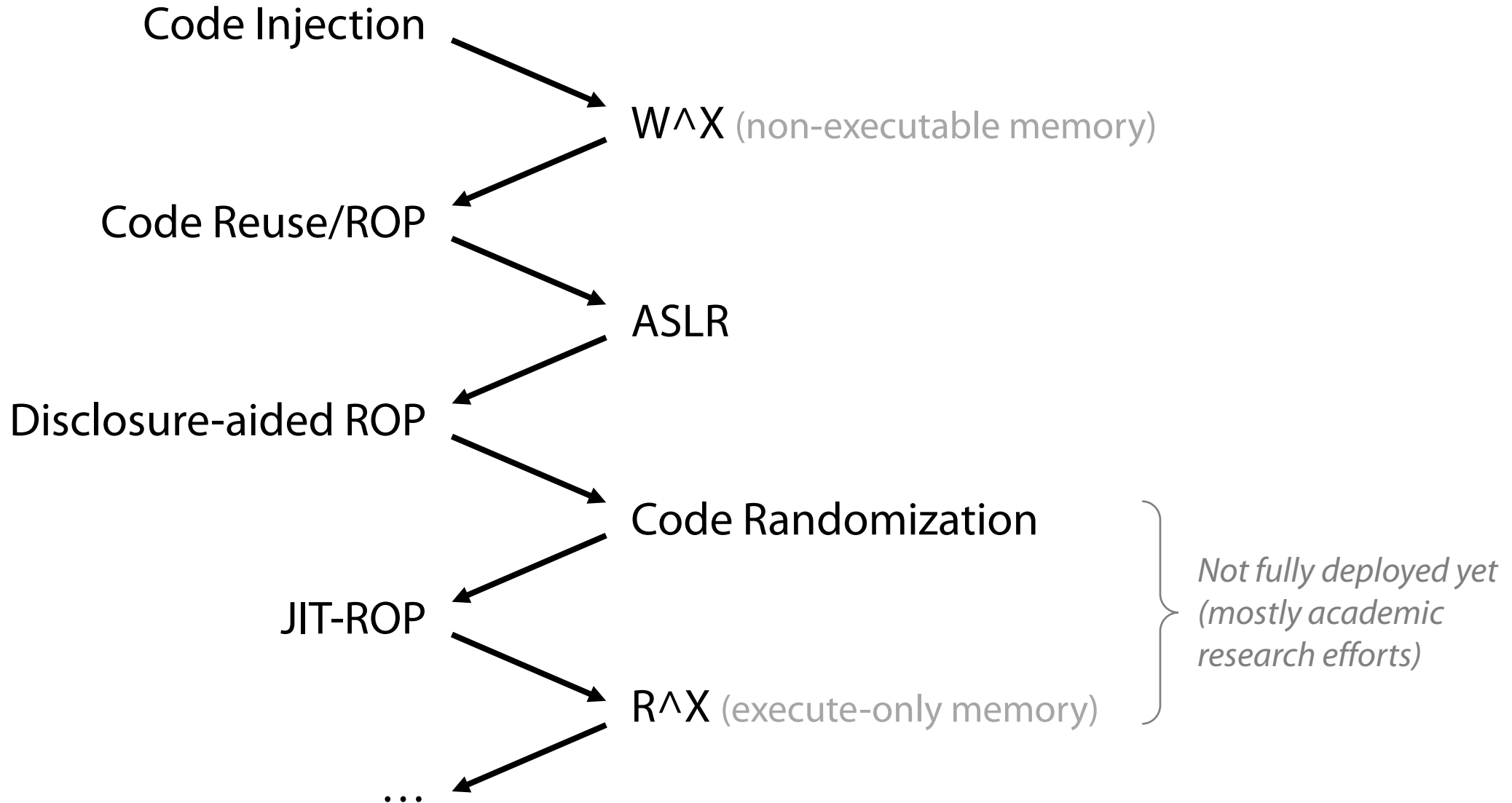
Some new CPUs support 3 bits per page: **R, W, X**

Before, any mapped page was implicitly *readable*

Advanced code reuse attacks rely on reading a process' code before executing it

R^X: Marking a code page as executable but *non-readable* prevents memory reads but still permits instruction fetches → breaks JIT-ROP

Exploitation vs. Mitigation



Direct vs. Indirect Branches

Direct control flow instructions `jmp 0xABCD`

The target is denoted by the operand (immediate value or memory address)

Relative or absolute jump: static analysis can identify the target

Indirect control flow instructions `jmp eax`

The target is denoted by the value of a register or memory location

Actual target can be known only at runtime

Control flow hijacking is possible because the attacker can influence the target of *indirect* branches

Change them to point *anywhere* in executable memory

Injected shellcode, libc function, ROP gadget, ...

Control Flow Integrity

Under normal conditions, execution will flow only into legitimate program locations

Beginning of functions, beginning of basic blocks, return call sites

Attackers can redirect execution to arbitrary locations

CFI: restrict control flow to only legitimate paths

According to predetermined (statically computed) control flow graph

Match `jmp/call/ret` call sites to target destinations

Statically assign labels to all indirect jumps and their targets

Dynamically validate that the target label matches the jump site before transferring control → runtime checks

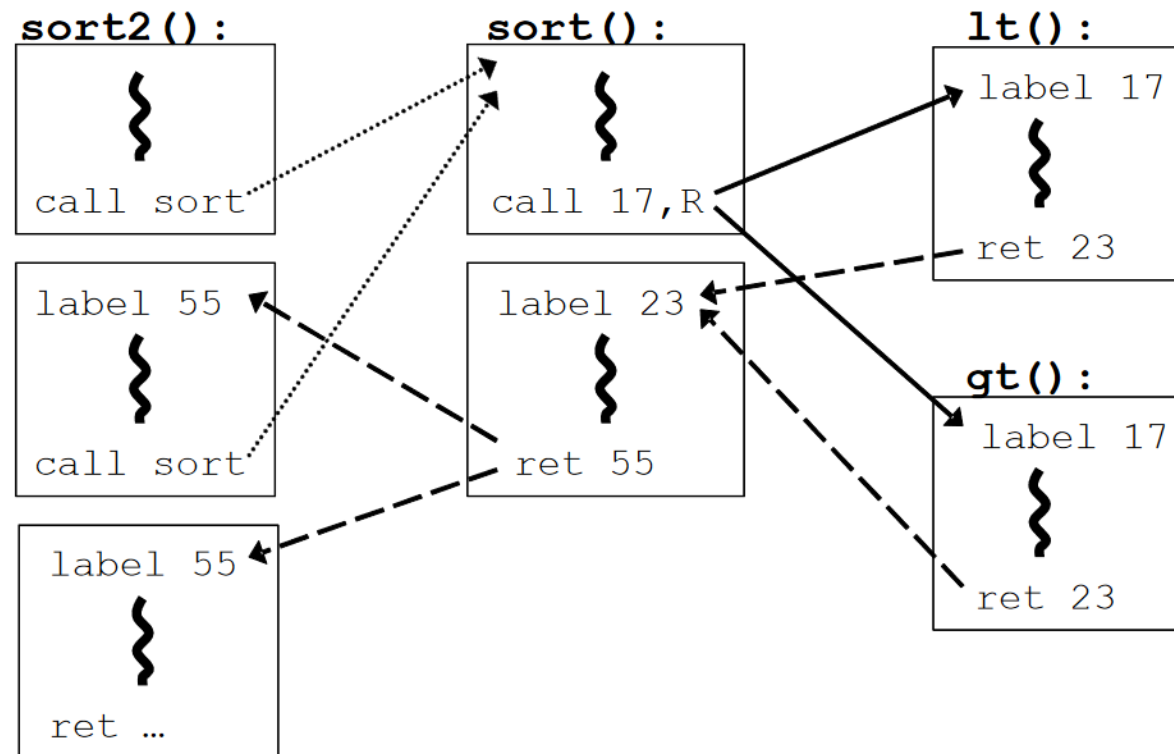
```

bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}

```



.....> Direct call
 —————> Indirect call
 ← - - - - Return

Example CFI instrumentation

Bytes	Instructions	Original Code
FF E1	jmp ecx	; computed jump
8B 44 24 04	mov eax, [esp+4]	; dst
...		

Instrumented Code

B8 77 56 34 12	mov eax, 12345677h	; load ID-1	3E 0F 18 05	prefetchnta	; label
40	inc eax	; add 1 for ID	78 56 34 12	[12345678h]	; ID
39 41 04	cmp [ecx+4], eax	; compare w/dst	8B 44 24 04	mov eax, [esp+4]	; dst
75 13	jne error_label	; if != fail	...		
FF E1	jmp ecx	; jump to label			

Jump to target address only if the tag at the destination == 0x12345678

Repurpose a no-sideeffect x86 instruction to hold the tag

Forward vs. Backward CFG Edges

Forward edge

- Function pointers

- Virtual calls

Backward edge

- Return instructions

Static analysis is not effective for return instructions

- Target set would include every possible call site for a given function

- Too large set to provide meaningful protection

Illegal Returns

Legitimate code:

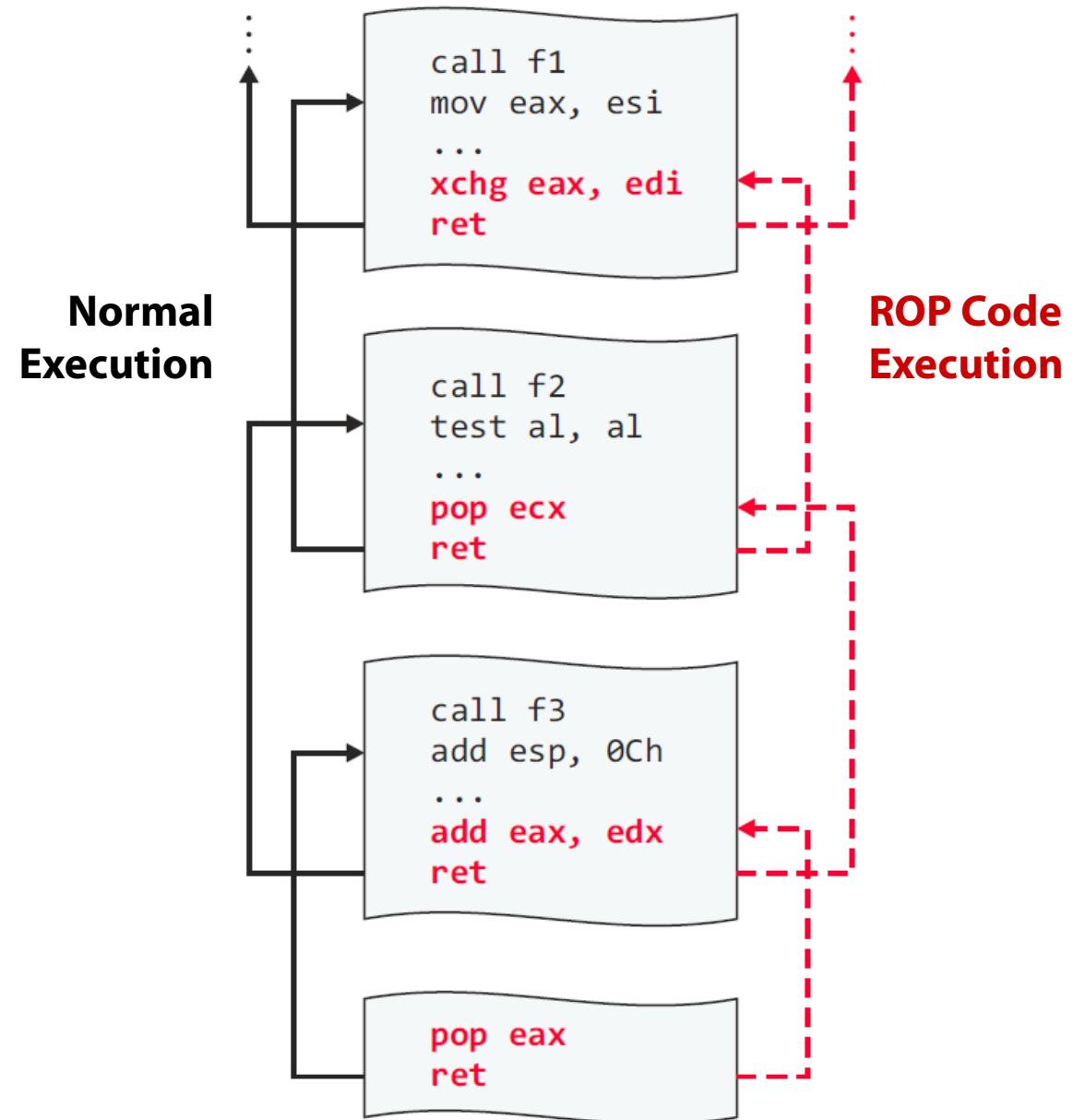
ret transfers control to the instruction right after the corresponding call

→ legitimate call sites

ROP code:

ret transfers control to the first instruction of the next gadget

→ arbitrary locations



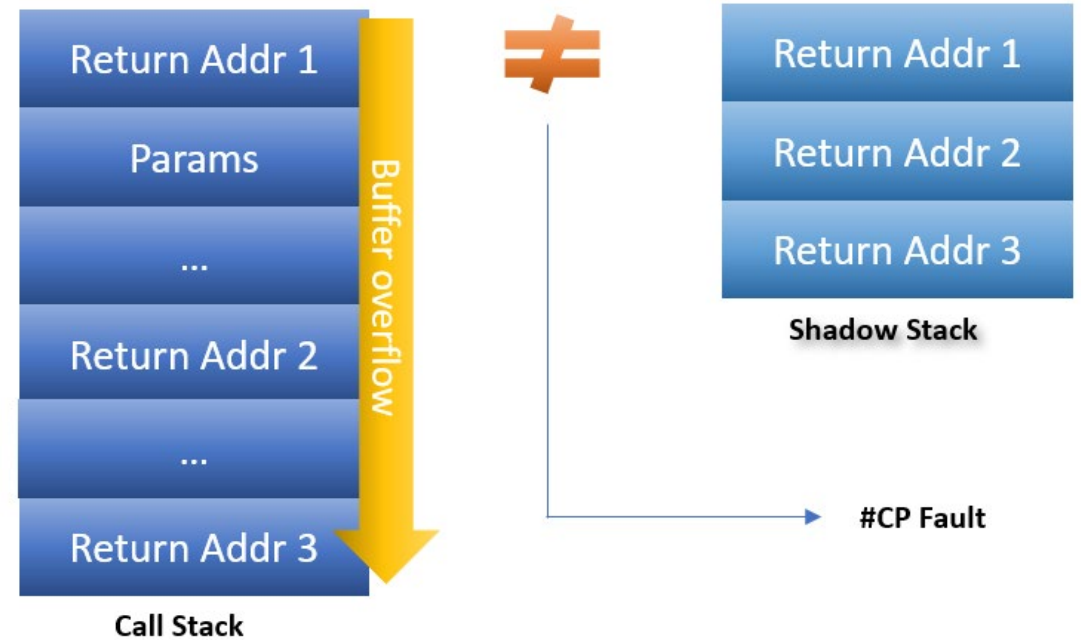
Shadow Stack

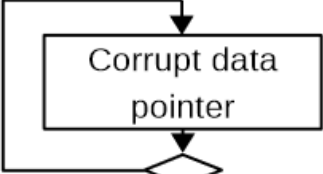
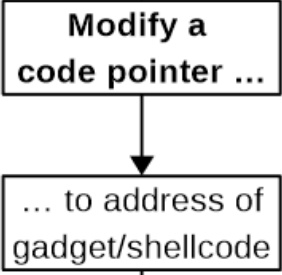
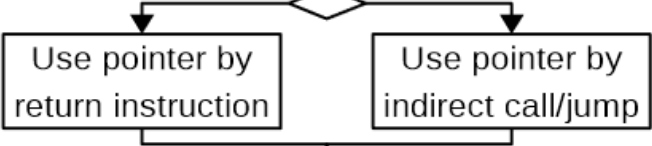

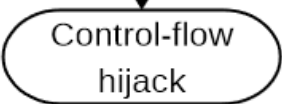
Separate return addresses from the data plane

Provide integrity protection for return addresses

Dynamic defense – does not rely on CFG analysis

Windows 10 introduced hardware-enforced stack protection



Attack step	Property	Mechanism	Stops all control-flow hijacks?	Avg. overhead
	Memory Safety	SoftBound+CETS [34, 35] BBC [4], LBC [20], ASAN [43], WIT [3]	Yes No: sub-objects, reads not protected No: protects red zones only No: over-approximate valid sets	116% 110% 23% 7%
	Code-Pointer Integrity (this work)	CPI CPS Safe Stack	Yes No: valid code ptrs. interchangeable No: precise return protection only	8.4% 1.9% ~0%
	Randomization	ASLR [40], ASLP [26] PointGuard [13] DSR [6] NOP insertion [21]	No: vulnerable to information leaks No: vulnerable to information leaks No: vulnerable to information leaks No: vulnerable to information leaks	~10% 10% 20% 2%
	Control-Flow Integrity	Stack cookies CFI [1] WIT (CFI part) [3] DFI [10]	No: probabilistic return protection only No: over-approximate valid sets No: over-approximate valid sets No: over-approximate valid sets	~2% 20% 7% 104%
	Non-Executable Data	HW (NX bit) SW (Exec Shield, PaX)	No: code reuse attacks No: code reuse attacks	0% few %
	High-level policies	Sandboxing (SFI) ACLs Capabilities	Isolation only Isolation only Isolation only	varies varies varies
				
				

Windows 10 Exploit Mitigations (non-exhaustive list)

Data Execution Prevention (DEP): non-executable memory

Structured Exception Handling Overwrite Protection (SEHOP): prevents SEH overwrite

Address Space Layout Randomization (ASLR): loads DLLs into random memory addresses

Heap protections: metadata hardening, allocation randomization, guard pages

Control Flow Guard (CFG): CFI for indirect calls

Shadow Stack (CET): Separate stack for return addresses

Code Integrity Guard (CIG): Images must be signed and arbitrary images cannot be loaded

Arbitrary Code Guard (ACG): Prevent dynamic code generation/modification/execution

Windows Defender Application Guard (WDAG): application sandboxing

Windows Defender SmartScreen: check the reputation of downloaded applications