IEEE SecDev | 2021

IEEE COMPUTER SOCIETY

Tutorial: Investigating Advanced Exploits for System Security Assurance

Daphne Yao
Professor
Virginia Tech

Salman Ahmed
PhD Candidate
Virginia Tech

Long Cheng
Assistant Professor
Clemson University

Hans Liljestrand
Postdoctoral Fellow
University of Waterloo

N. Asokan
Professor
University of Waterloo

#IEEESecDev https://secdev.ieee.org/2021

VIRGINIA TECH CLEMSON UNIVERSITY UNIVERSITY OF WATERLOO

Purposes of this Tutorial

To help understand **advanced attack/defense** techniques with **hands on activities**

To inspire promising **defense and measurement opportunities** in system security

The need for breaking down advanced exploits

Attack investigation can provide us insights on:

measurable metrics

systemic measurement methodologies

} Necessary for system security assurance

Assessing impact of defenses on attack components.

choosing effective security parameters.

Improving awareness on system security.

Will discuss many system security topics:

1. Data-oriented attacks and their defenses
2. Leaked addresses or pointers
3. Time in exploits
4. Defense schemes (e.g., block vs instruction-level randomization)
5. Hardware-assisted protections

In our CCS 2020 work[1], we find out:

1. Attackers only need several seconds to find Turing Complete gadgets
2. Locations of leaked addresses/pointers have no impact on gadget availability, but affect how fast attackers find gadgets
3. Instruction-level single-round randomization still works under JIT-ROP!

Details of these impact will be covered in later slides

[1]Salman Ahmed, Ya Xiao, Kevin Z. Snow, Gang Tan, Fabian Monrose, and Danfeng (Daphne) Yao. 2020. Methodologies for Quantifying (Re-)randomization Security and Timing under JIT-ROP. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), 1803–1820. DOI:https://doi.org/10.1145/3372297.3417248

FEATURE

The Microsoft Exchange Server hack: A timeline

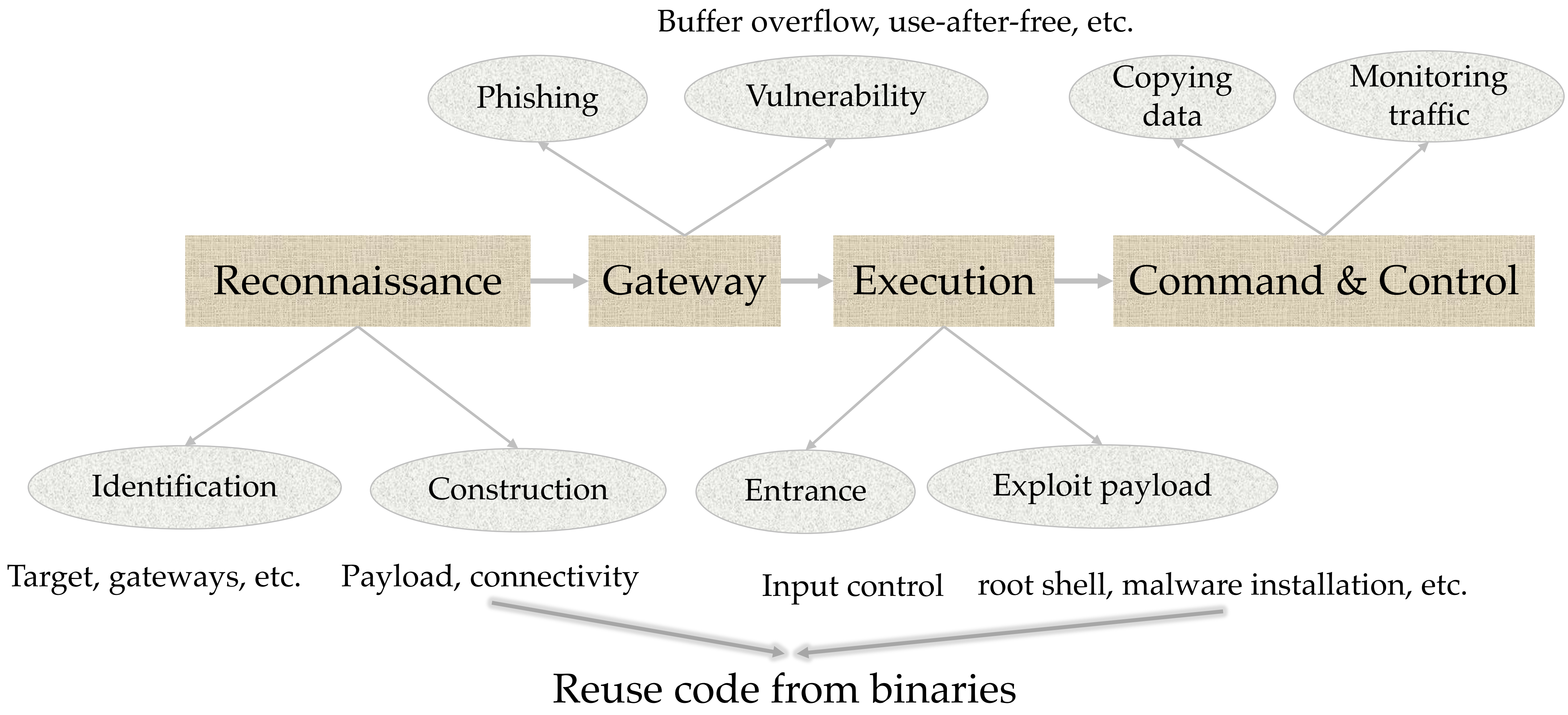
Research shows plenty of unpatched systems remain. Here's how the attacks unfolded, from discovery of vulnerabilities to today's battle to close the holes.

Improving the exploit for CVE-2021-26708 in the Linux kernel to bypass LKRG

Aug 25, 2021

This is the follow-up to my research described in the article "Four Bytes of Power: Exploiting CVE-2021-26708 in the Linux kernel." My PoC exploit for CVE-2021-26708 had a very limited facility for privilege escalation, and I decided to continue my experiments with that vulnerability. This article describes how I improved the exploit, added a full-power ROP chain, and implemented a new method of bypassing the Linux Kernel Runtime Guard (LKRG).

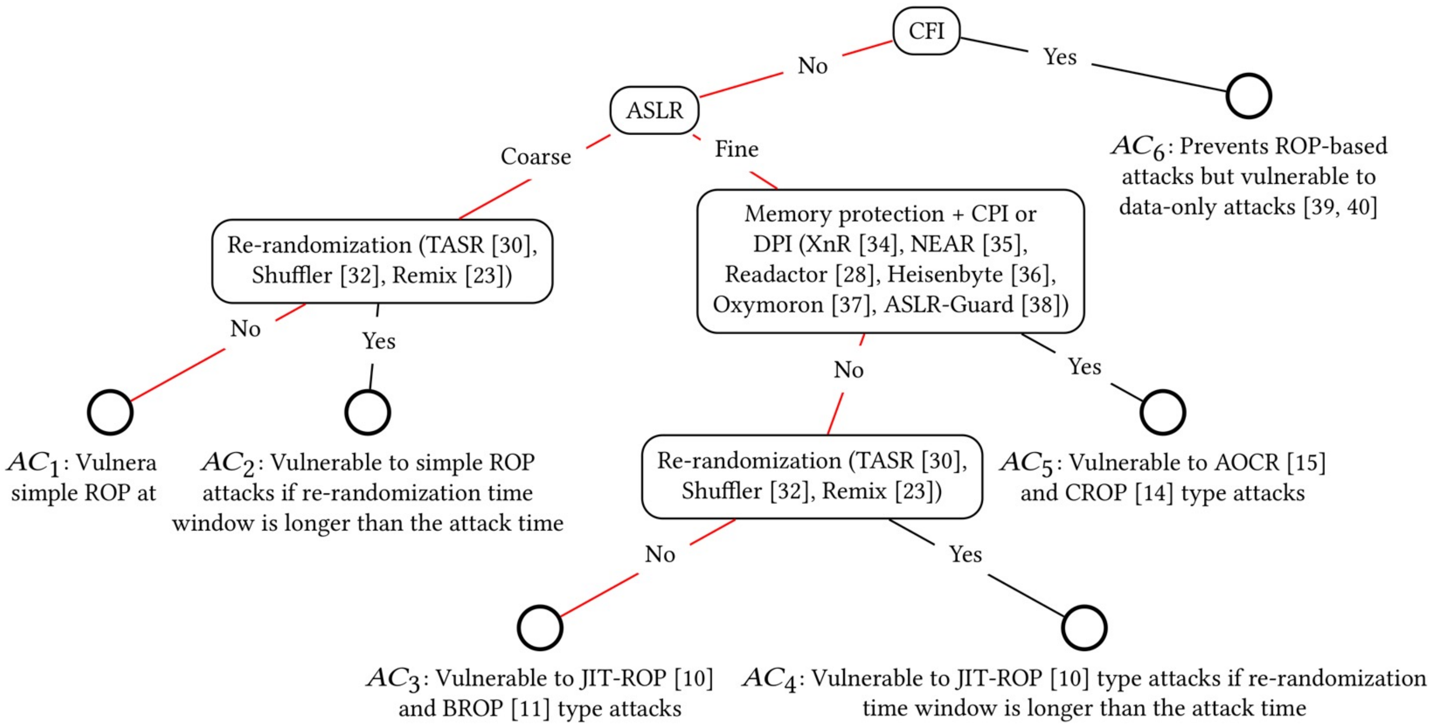
Multiple Phases of an Exploit



Factors of a Successful Exploit

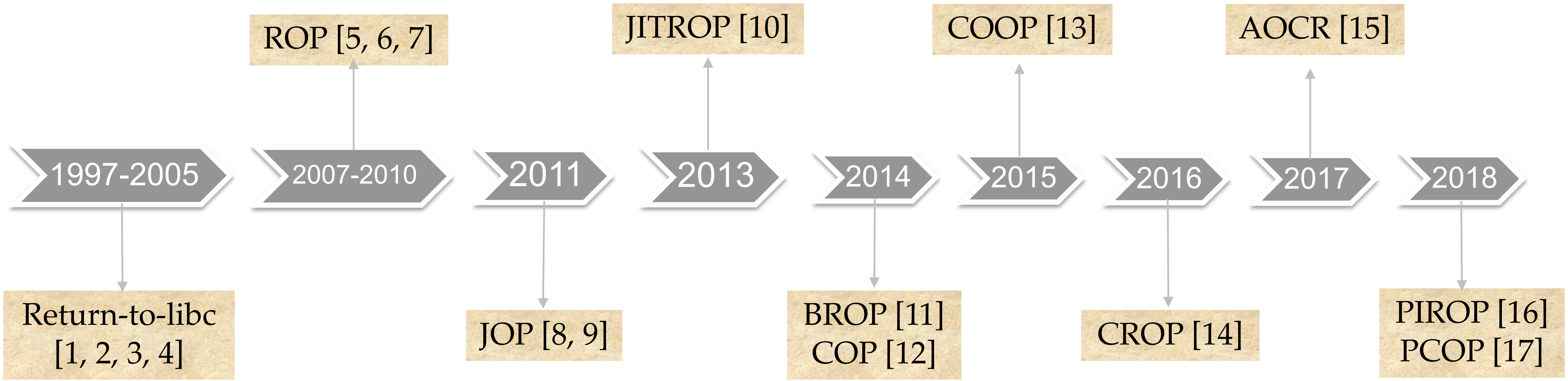
Reconnaissance must consider the underlying defenses in the system

- Memory disclosure is necessary for code reuse attacks.
  - code pointer leak
  - object pointer leak
- Availability of reusable code and its quality (i.e., gadget quality)
- Availability of system interfaces (i.e., system calls)
- Triggerable vulnerability
- Gadget reachability



Presence of defenses lead to different attack conditions

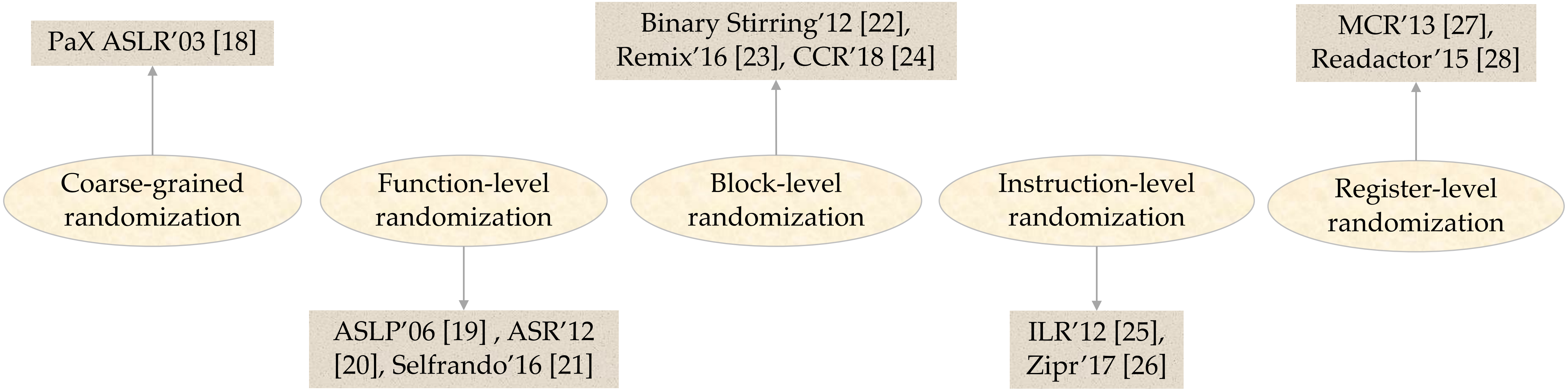
History of Code Reuse Attacks



ROP: Return-Oriented Programming
JOP: Jump-Oriented Programming
JITROP: Just-In-Time Return-Oriented Programming
BROP: Blind Return-Oriented Programming
PIROP: Position Independent ROP

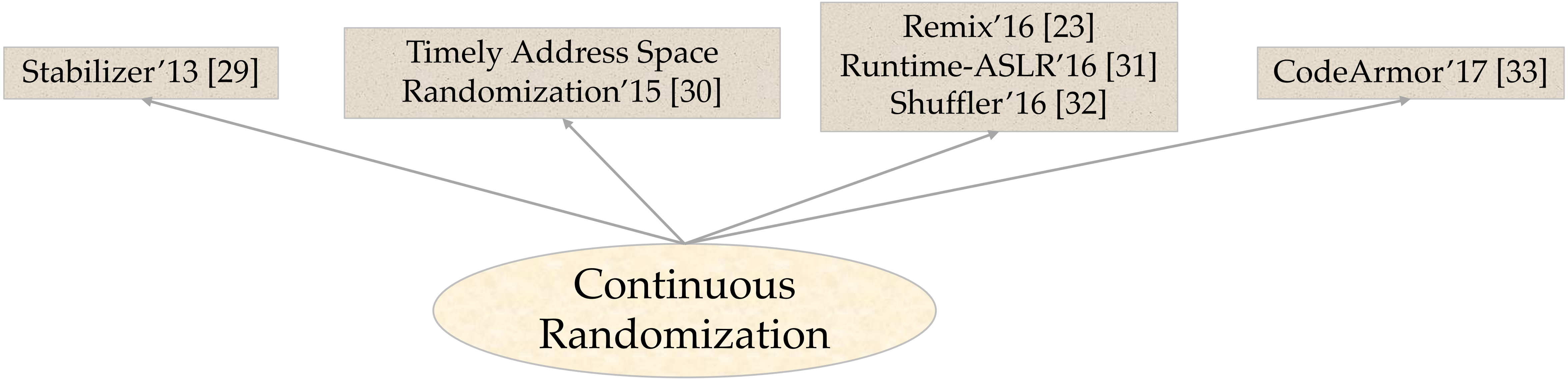
AOCR: Address Oblivious Code Reuse
COP: Call-Oriented Programming
COOP: Counterfeit Object-Oriented Programming
CROP: Crash-Resistance Oriented Programming

History of Memory Randomization (1)



ASLR: Address Space Layout Randomization
ASLP: Address Space Layout Permutation
ASR: Address Space Randomization
CCR: Compiler-assisted Code Randomization
MCR: Multicompiler

History of Memory Randomization (2)

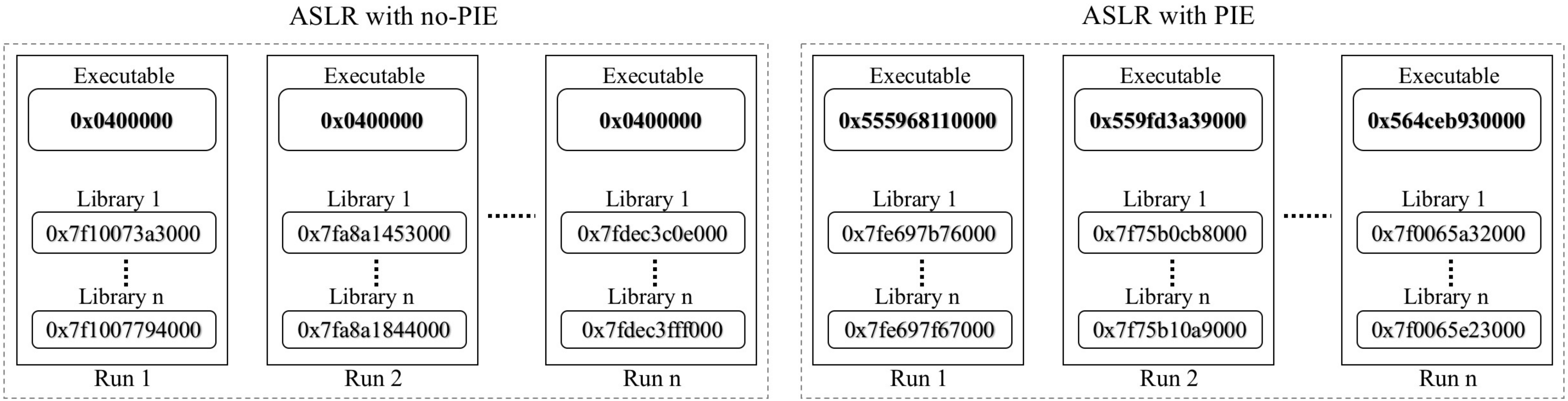


Latest versions of Windows, Linux, MacOS, Android, and iOS operating systems support only the coarse-grained ASLR with Position Independent Executable (PIE).

ASLR (aka Coarse-grained ASLR)

Makes the finding of gadgets in known addresses (i.e., code reuse) difficult – attackers still able to deduce gadgets from leaks.

Position Independent Executable (PIE) extends ASLR to randomize address of main binary on each run.



ASLR + PIE

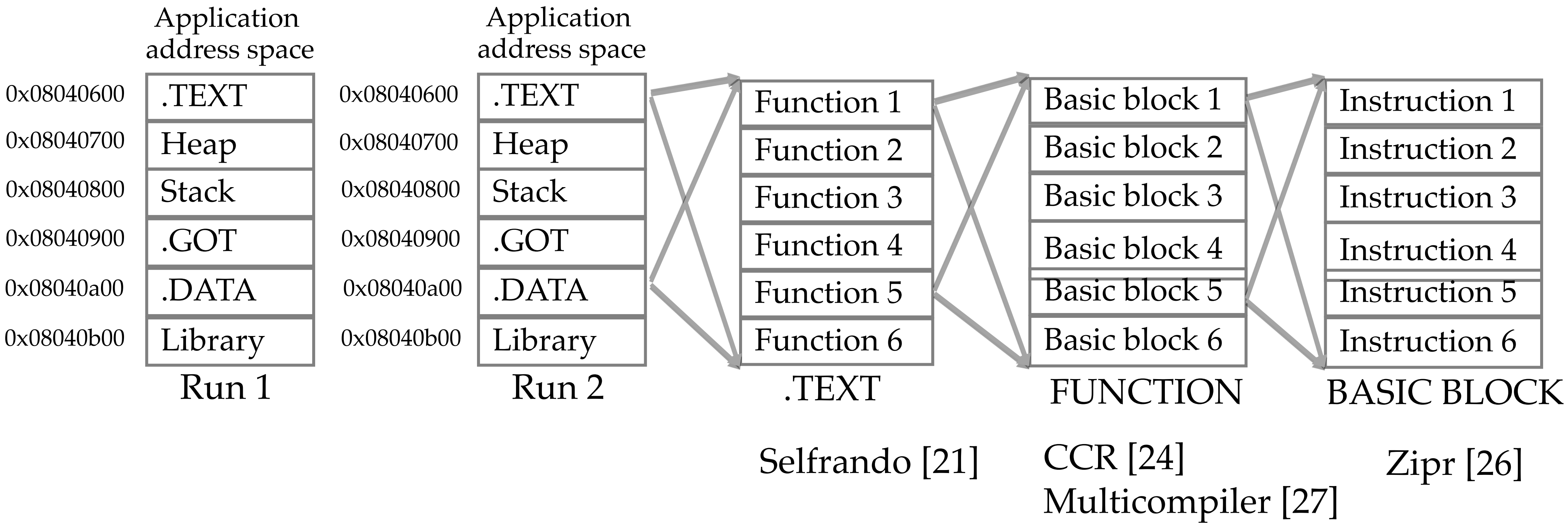
Address Space Layout Randomization or ASLR aims to make the code reuse task difficult by randomizing the location of functions or gadgets.

Application address space

Address	Run 1	Run 2	Run 3
0x08040600	.TEXT	.TEXT	Heap
0x08040700	Heap	Heap	.DATA
0x08040800	Stack	Stack	.TEXT
0x08040900	.GOT	.GOT	Library
0x08040a00	.DATA	.DATA	.GOT
0x08040b00	Library	Library	Stack

Fine-grained ASLR

Coarse-grained ASLR may not be effective in case of leaks (e.g., code pointer leaks, object pointer leaks, etc.).



Speakers’ Component in Our Tutorial Today

Daphne Yao

1. Overview of advanced attacks and various defenses.

Salman Ahmed

2. Code reuse attacks, ROP, ASLR, JITROP, and Demonstrations.

Long Cheng

3. Overview of data-oriented attacks using data manipulation.

Hans Liljestrand

4. Demonstration of DOP exploits and defenses.

N. Asokan

5. Research directions in hardware-assisted protection

Long Cheng

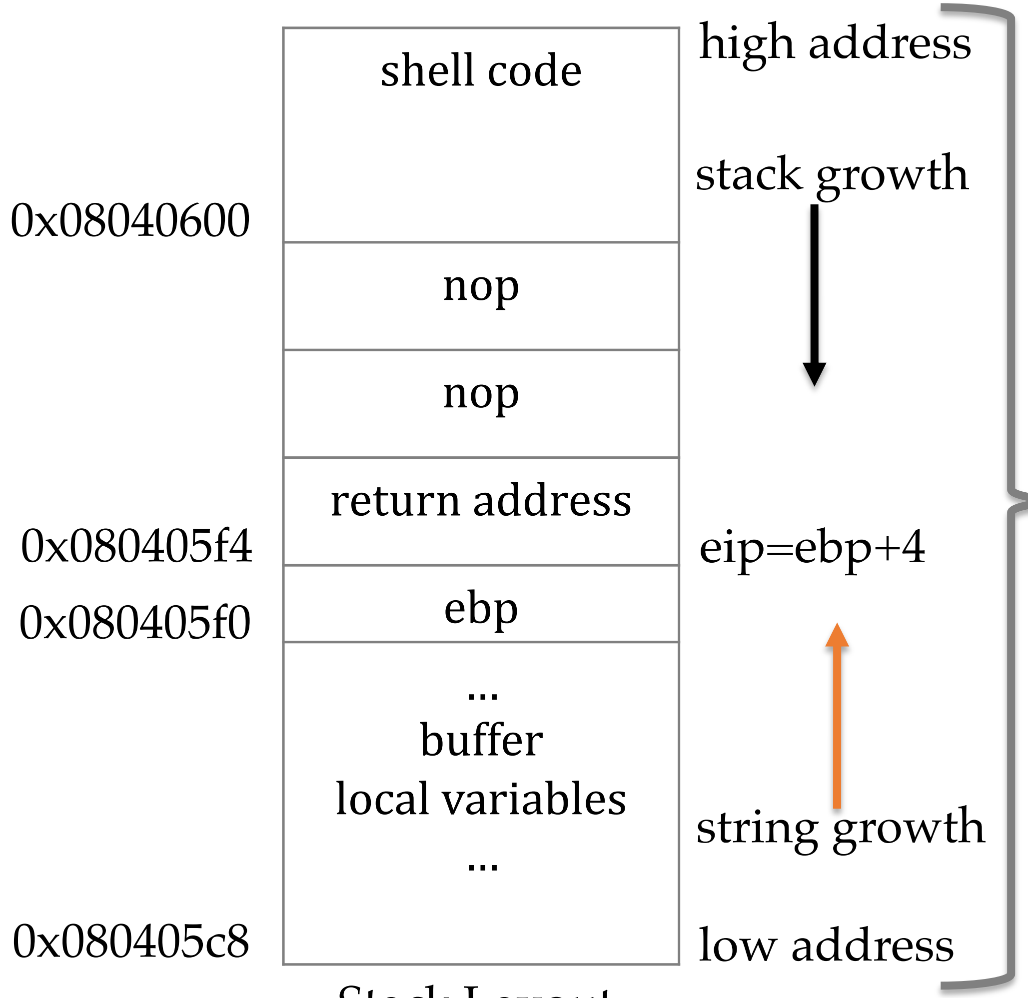
6. Concluding remarks and research directions.

VIRGINIA TECH. CLEMSON UNIVERSITY UNIVERSITY OF WATERLOO

IEEE SecDev|2021

- Code Reuse Attacks,
- Return-Oriented Programming (ROP),
- Just-In Time ROP (JITROP), and
- Demonstration

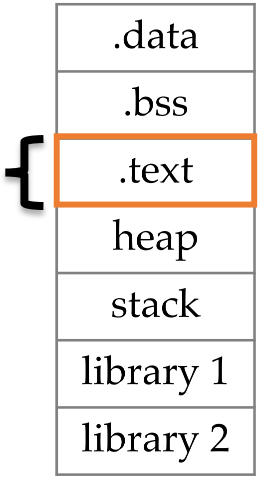
Code Reuse Attack



Stack Layout

Injecting shell code in stack is prohibited by the **NX** or **DEP** because stack is not executable.

Code reuse bypasses DEP or NX by constructing shell code using existing code, i.e., without injecting anything.



Binary Layout

Code-reuse technique constructs shell code using whole functions or gadgets

system()
exit()

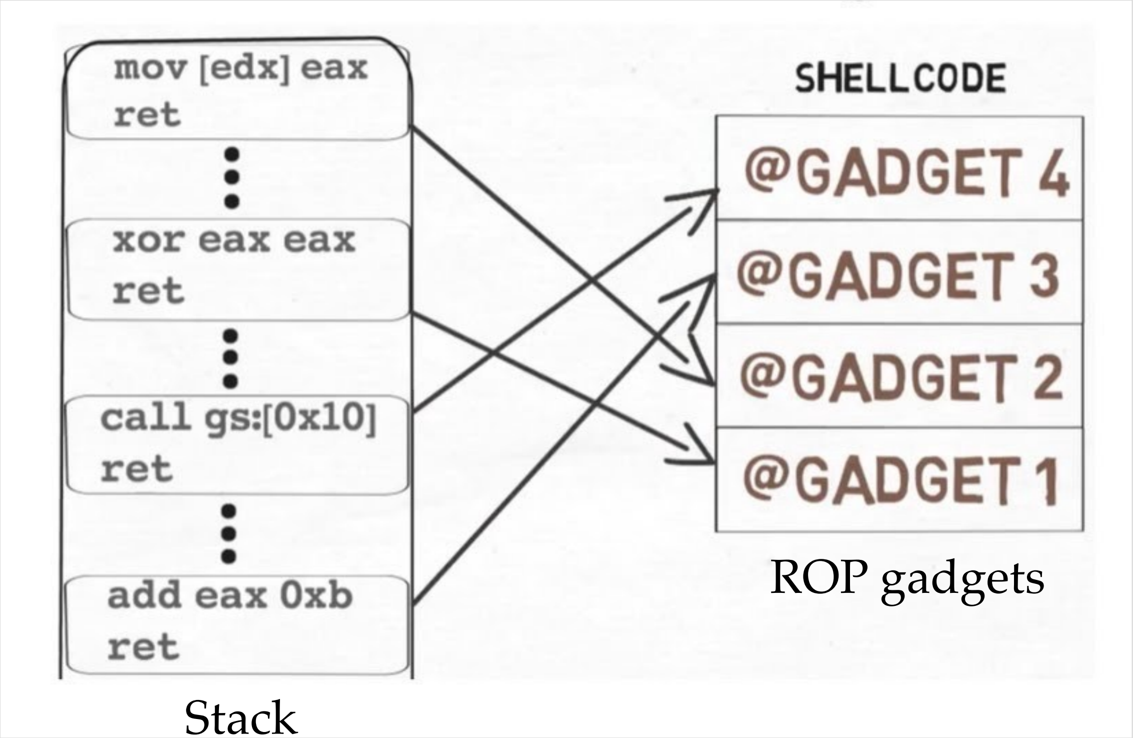
Whole functions

0x08080A24: pop eax; ret
0x08080C46: pop ebx; ret
0x0808A058: mov [ebx], eax; ret

Gadgets

Return-Oriented Programming (ROP) [5]

ROP uses short instructions followed by ret. These short instruction sequences are called gadgets. Each gadget has a specific purpose.



Chaining gadgets to achieve a malicious goal.

ROP Gadgets can Achieve Turing-complete (TC) operations [42]

Gadget types	Purpose	Minimum footprint	Example	TC?
Move register	Sets the value of one register by another	mov reg1, reg2; ret	mov rdi, rax; ret	✓
Load register	Loads a constant value to a register	pop reg; ret	pop rbx; ret	✓
Arithmetic	Stores an arithmetic operation's result of two register values to the first	△ reg1, reg2; ret	add rcx, rbx; ret	✓
Load memory	Loads a memory content to a register	mov reg1, [reg2]; ret	mov rax, [rdx]; ret	✓
Arithmetic load	△ a memory content to/from/by a register and store in that register	△ reg1, [reg2]; ret	add rsi, [rbp]; ret	✓
Store memory	Stores the value of a register in memory	mov [reg1], reg2; ret	mov [rdi], rax; ret	✓
Arithmetic store	△ a register value to/from/by a memory content and stores in that memory	△ [reg1], reg2; ret	sub [ebx], eax; ret	✓
Logical	Performs logical operations	ϕ reg1, reg2; ret ϕ reg1, const; ret ϕ [reg1], reg2; ret ϕ [reg1], const; ret	shl rax, cl; ret;	✓
Stack pivot	Sets the stack pointer, SP	∇ sp, reg	xchg rsp, rax	×
Jump	Sets instruction pointer, EIP.	jmp reg	jmp rdi	✓
Call	Jumps to a function through a register or memory indirect call	call reg or call [reg]	call rdi	✓
System Call	Invokes system functions	syscall or int 0x80; ret	syscall	✓
Call preceded	Bypasses call-ret ROP defense policy	mov [reg1], reg2; call reg3	mov [rsp], rsi; call rdi	×
Context switch	Allows processes to write to Last Branch Record (LBR) to flash it	long loop.	3dd4: dec, ecx 3dd5: fmul, [BC8h] 3ddb: jne, 3dd4	×
Flashing	Clears the history of LBR (Last Branch Record)	Any simple call preceded gadgets with a ret instruction	jmp A ... A: mov rax, 3; ret;	×
		Any gadgets that are		

Terminal	Bypasses kBouncer heuristics	Any gadgets that are 20 instructions long	N/A	×
Reflector	Allows to jump to both call-preceded or non-call-preceded gadgets	mov [reg1], reg2; call reg3; ... ; jmp reg4	mov [rsp], rsi; call rdi; ... ; jmp rax	×
Call site	This gadget chains the control to go forward when we have the control on the stack and ret	call reg or call [reg]; ... ret;	call rdi; ... ret;	×
Entry point	This gadget chains the control to go forward when we have the control of a call instruction	pop rbp; ... call/jmp reg or call/jmp [reg]	pop rbp ... call/jmp reg or call/jmp [reg]	×
BROP	Restores all saved registers	pop rbx; pop rbp; pop r12; pop r13; pop r14; pop rsi; pop r15; pop rdi; ret;	pop rbx; pop rbp; pop r12; pop r13; pop r14; pop rsi; pop r15; pop rdi; ret;	×
Stop	Halts the program execution	Infinite loop	4a833dd4: inc rax 3ddb: jmp 3dd4	×

We compiled various gadgets from multiple sources [41].

Other gadget categories include MOV TC, priority, and payload gadget sets [41].

Real-World Code Reuse Attacks

Thursday, August 6, 2015

One font vulnerability to rule them all #2: Adobe Reader RCE exploitation

Posted by Mateusz Jurczyk of Google Project Zero

Thursday, August 13, 2015

One font vulnerability to rule them all #3: Windows 8.1 32-bit sandbox escape exploitation

Posted by Mateusz Jurczyk of Google Project Zero

Details of one font exploitation in the next slide.

Exploit of One Font Vulnerability

Vulnerability	Reason	Affected programs	Mitigation bypasses
CVE-2015-0093	unlimited out-of-bounds stack manipulation	Adobe Reader 11.0.10 on Windows 8.1 Update 1, both 32-bit and 64-bit.	Stack cookies, DEP, ASLR, and SMEP

Technique	ROP gadgets	System functions
- Stack pointer (SP) manipulation - Manipulation through charstring program - ROP gadgets - System functions	XCHG EAX, EDX MOV EBX, EDX POP ESI POP ECX REP MOVSD JMP EBX	VirtualProtect GetProcAddress(), LoadLibrary() NtGdiAddRemoteFontToDC

Also, allows elevation of privileges in the Windows kernel through processes.

- Coarse-grained ASLR

  Key Limitation: Can be bypassed using information leaks

- Fine-gained ASLR

  Goal: aims to protect information leaks

Does then fine-grained ASLR make code reuse attacks impossible?

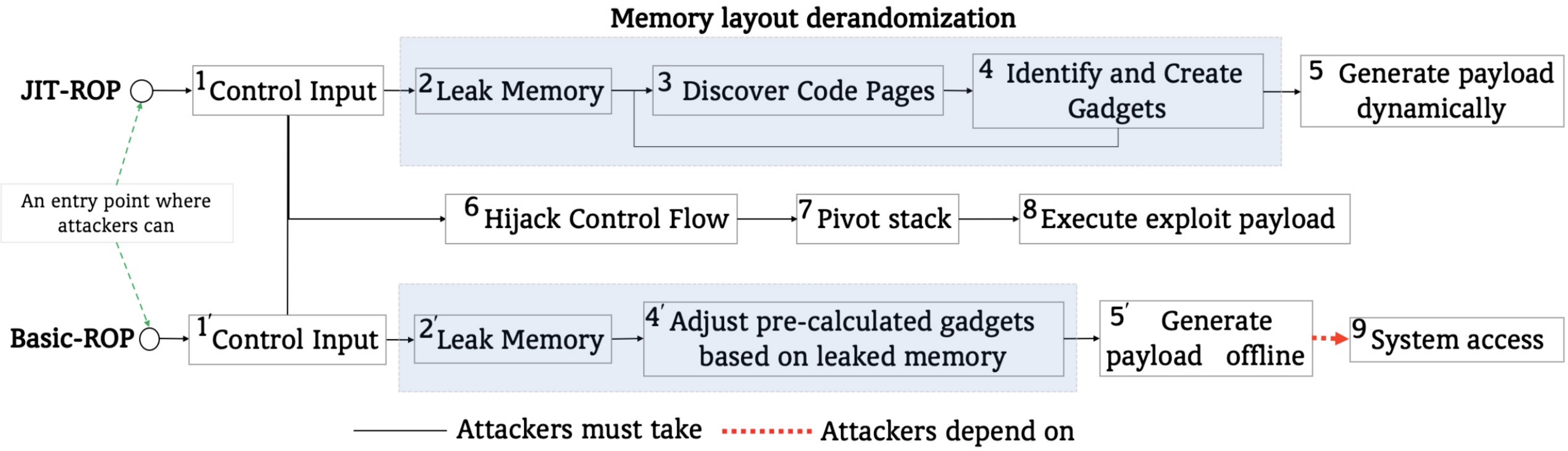
No!!!

Just-In-Time Return-Oriented Programming[2] (JITROP) [10]

[2]Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In 2013 IEEE Symposium on Security and Privacy, pages 574–588. IEEE, 2013.

Just-In-Time ROP or JITROP [10]

The key difference between ROP and JITROP is **how the gadget is discovered**. JITROP **dynamically** discovers the gadgets.

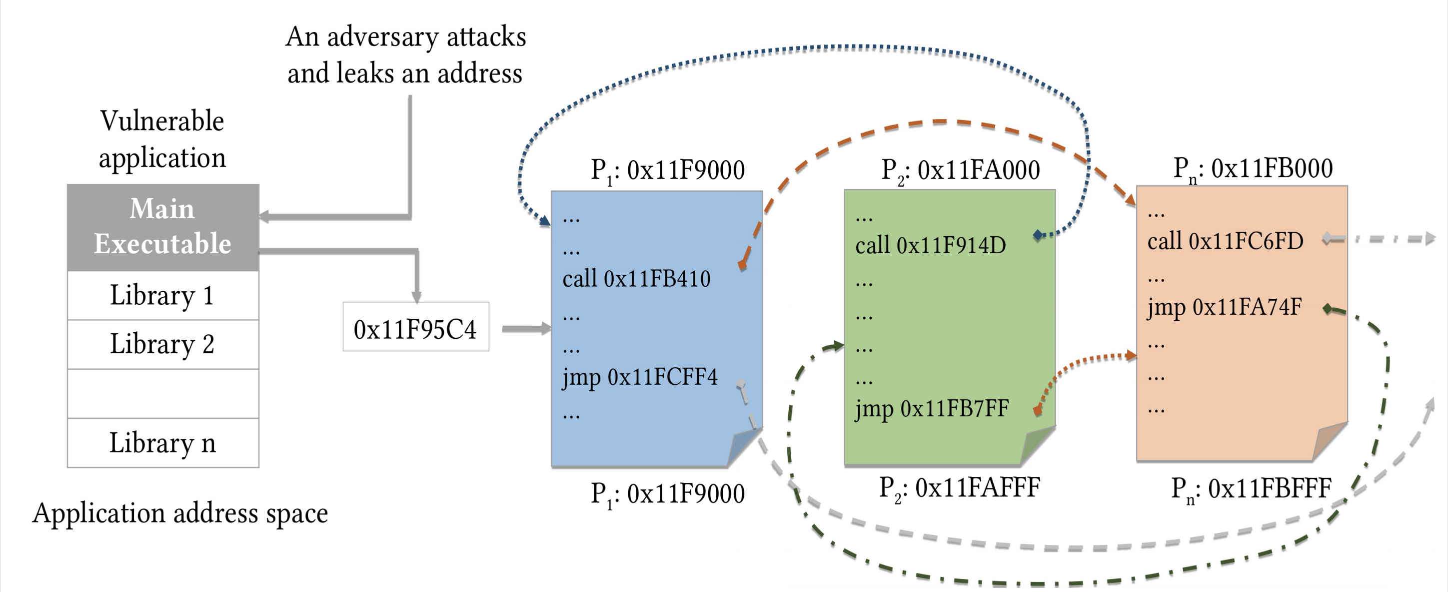


JITROP's Dynamic Code Harvest

JITROP's uses dynamic code harvesting technique to discover ROP gadgets.

The code harvesting starts from a single code address/pointer leak

The technique leaks repeatedly leaks multiple code pointers from the single leak.



P_1, P_2, … P_n are 4 KB code pages

JITROP's Requirements and Some In-depth Questions

JITROP [10] is a powerful attack technique known for bypassing fine-grained ASLR. But it requires a code address/pointer leak to start with.

Also, some in-depth questions require answer:

1) How much **time** can an attack have to perform JIT-ROP attacks considering different expressiveness of ROP attacks?

2) What impact do **fine-grained ASLR schemes** have on the Turing-complete expressiveness of JIT-ROP payloads?

3) How do attack vectors (e.g., **starting code pointer leaks**) impact the JIT-ROP attacks?

We have addressed these in-depth questions in our work titled

“Methodologies for Quantifying (Re-)randomization Security and Timing under JIT-ROP*” [41]

*Salman Ahmed, Ya Xiao, Kevin Z. Snow, Gang Tan, Fabian Monrose, and Danfeng (Daphne) Yao. 2020. Methodologies for Quantifying (Re-)randomization Security and Timing under JIT-ROP. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS ‘20), 1803–1820. DOI:https://doi.org/10.1145/3372297.3417248

Attackers Require a few Seconds!

The upper bound* ranges from **1.5 to 3.5 seconds** in our tested **17** applications such as nginx, proftpd, firefox, etc with **FOUR** gadget sets [41].

	Time to leak all gadget types	
Gadget set	**Minimum (s)**	**Average (s)**
TC	2.2	4.3
Priority	1.5	3.5
MOV TC	3.5	5.3
Payload*	2.1	4.8
Average	**2.3s**	**4.5s**

* May vary with machine configurations

Impact of Fine-grained ASLR Schemes

Single-round **instruction-level** randomization limits up to **90%** gadgets [41] and restricts Turing-complete operations.

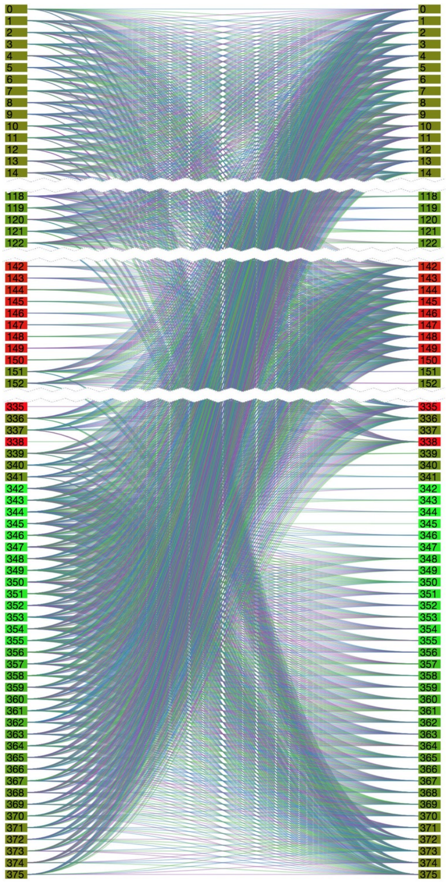
So, instruction-level randomization is still useful.

Randomization schemes	Granularity	⇓ (%) MIN-FP	⇓ (%) EX-FP
Main executables			
Inst. level rando. [50]	Inst.	**79.7**	**82.5**
Func. level rando. [25]	FB	27.63	36.55
Func.+Reg. level rando. [53]	FB & Reg.	17.62	42.37
Block level rand. [59]	BB	19.58	44.64
Dynamic libraries			
Inst. level rando. [50]	Inst.	**81.3**	**92.2**
Func. level rando. [25]	FB	46.5	43.8
Func.+Reg. level rando. [53]	FB & Reg.	44.2	43.9
Block level rand. [59]	BB	20.98	37.0

Reduction of Turing-complete gadget set with different randomization schemes

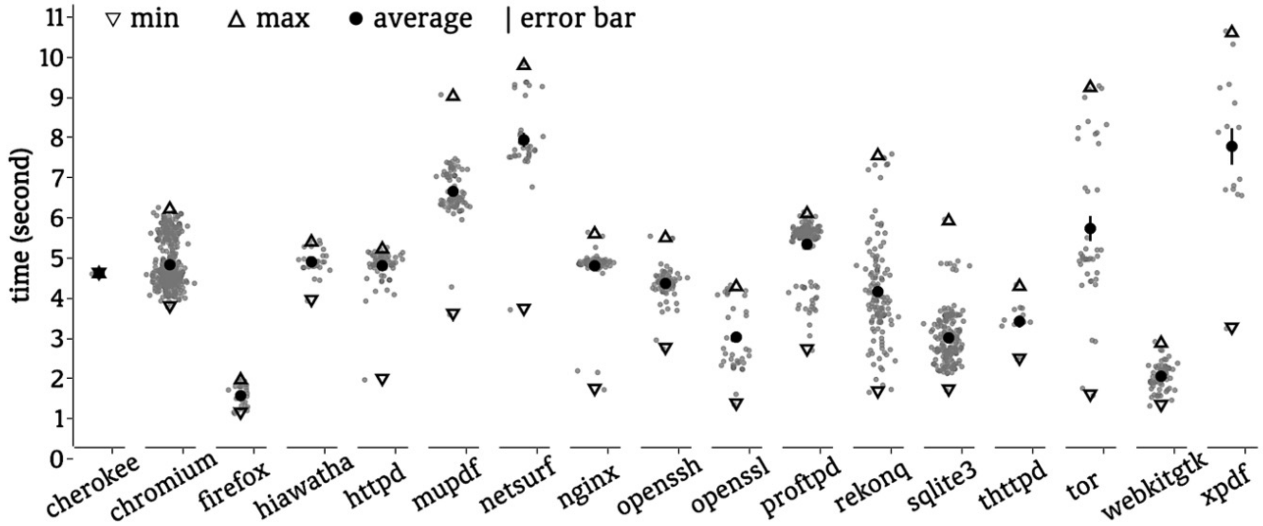
Impact of the Location of Pointer Leakage

No impact on connectivity



Connectivity of libc

Has an impact on the attack time: dense code pages contain diverse set of gadgets



Impact of starting pointer locations on gadget harvesting time.

Is protecting code-reuse attacks (or in broader term control-oriented attacks) **impossible**?

No!!!

Control-Flow Integrity

Demo Time

Demo Setup

1. Download our tutorial repository from GitHub

    $ git clone https://github.com/salmanyam/tutorial-secdev-2021.git
    or download the repository as zipped and unzip it.

2. Install Docker if it is not already installed using the instructions in the following link

    https://docs.docker.com/engine/install/ubuntu/ or run docker-install.sh script given in our repo.
    $ ./docker-install.sh

3. Build a docker image using the provided Docker file in the tutorial repo. This may take 2-3 minutes to complete.

    $ cd tutorial-secdev-2021
    $ sudo docker build -t secdevt21 .

4. Run the docker image with privileged mode. The priviledged mode is necessary for ptrace that is used in gdb for attaching a process and in our gadget finding code.

    $ sudo docker run -it --privileged secdevt21

Gadget Lookup

1. Run the nginx program given in the tutorial rep. The following command will start nginx server and print a leaked address in the terminal.

    $ ./nginx -c nginx.conf -g 'daemon on;' -p nginx

2. Get the pid of the nginx master process

    $ ps aux | grep nginx

3. Give the following command to get the Turing-complete gadget set

    $ ./jitrop -p <pid> -a <address>

4. To get other gadget sets, add an operation flag the end of the previous command as follows for example.

    $ ./jitrop -p <pid> -a <address> -o 7 [7 for MOV TC gadget set]

Gadget Lookup Time

To get gadget lookup times, we can change the operation value as follows:

-o 1: Operation 1 outputs the time to collect all the gadgets from the Turing-complete gadget set.
-o 2: Operation 2 outputs the time to collect all the gadgets from the priority gadget set.
-o 3: Operation 3 outputs the time to collect all the gadgets from the MOV TC gadget set.
-o 5: Operation 5 outptus the time to collect all the gadgets from a payload gadget set.

For example, the following command gives times to get all gadgets from Turing-complete gadget set.

```
$ ./jitrop -p <pid> -a <address> -o 1
```

Impact of Different Starting Pointers on Gadget Lookup

0x08040800

0x08040000

Changing the 4th bit from LSB gives a new code page address

0x08041800	0x0804B800	0x08045800
0x08041000	0x0804B000	0x08045000

Run gadget look from different starting pointer location and observe the impact on

i) gadget availability and
ii) ii) gadget lookup time

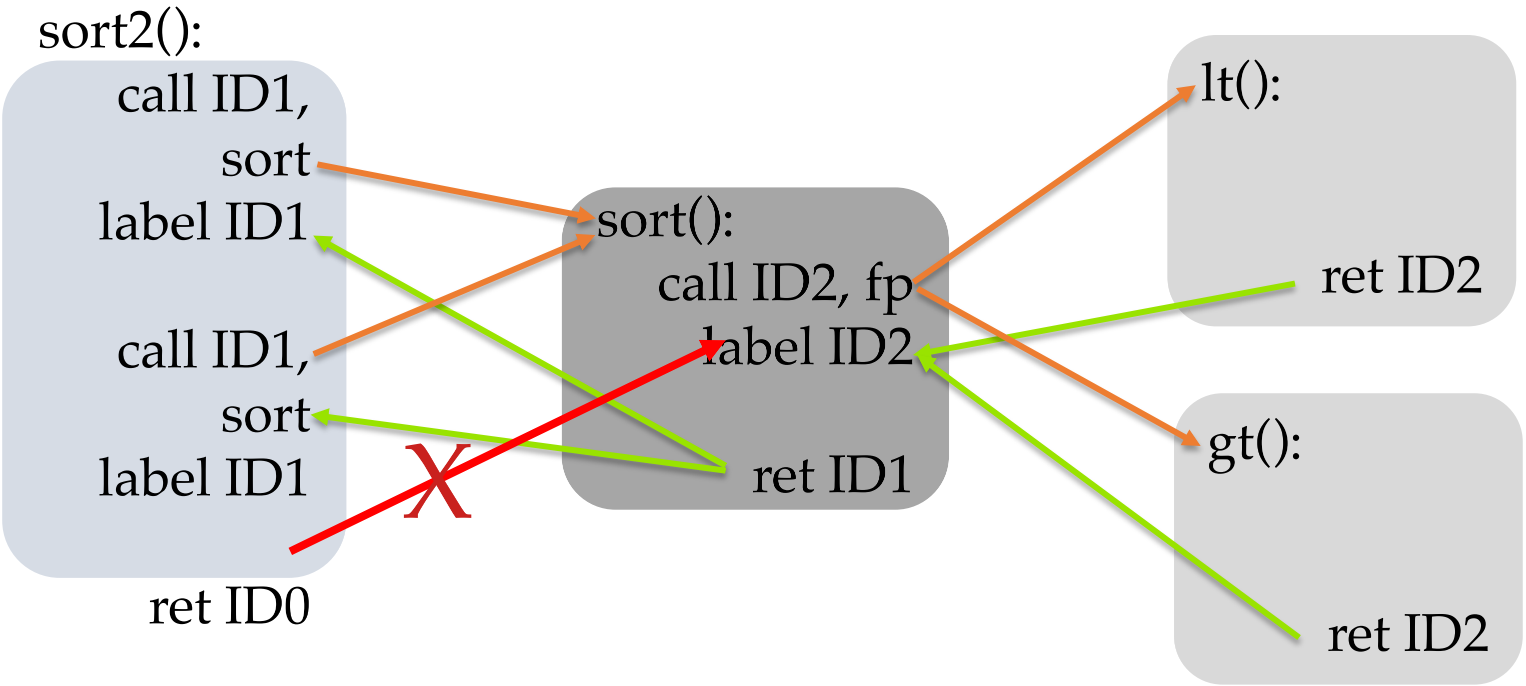
Control Flow Integrity (CFI)

CFI aims to provide strong protection against all control-oriented attacks.

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

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

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



Program can jump and return to only legitimate targets defined in control-flow graph.

Speakers' Component in Our Tutorial Today

Daphne Yao

1. Overview of advanced attacks and various defenses.

Salman Ahmed

2. Code reuse attacks, ROP, ASLR, JITROP, and Demonstrations.

Long Cheng

3. Overview of data-oriented attacks using data manipulation.

Hans Liljestrand

4. Demonstration of DOP exploits and defenses.

N. Asokan

5. Research directions in hardware-assisted protection

Long Cheng

6. Concluding remarks and research directions.

Memory Corruption Attacks

- Control-flow attacks
  - Increasingly difficult due to many deployed defenses

- Non-control data attacks (Data-oriented attacks)
  - An appealing attack technique
  - Without violating control-flow integrity

History of Data-Oriented Attacks

2005

Non-control-data attacks [Chen et al., *Security*]

2015

FlowStitch [Hu et al., *S&P*]

2016

Bypass SOP enforcement In Chrome [Jia. et al., CCS]

Data-Oriented Programing [Hu et al., *S&P*]

2017

Internet Explorer and Chrome [Rogowski. et al., Euro *S&P*]

Undermine CFI protection [Davi. et al., NDSS]

2018

Block-Oriented Programming [Ispoglou et al., CCS]

Nginx and Apache web servers [Morton. et al., Euro *S&P*]

Classification of Data-Oriented Attacks

- Direct data manipulation (DDM)
  - Directly manipulate the target data

- Data-oriented programming (DOP)
  - Abuse existing short code sequences, and re-engineer them for malicious purposes
  - Indirectly manipulate the target data
  - BOP (Block-Oriented Programming)

DDM Example

➢Format string vulnerability, buffer overflow, and double free vulnerabilities, etc.

```
pw->pw_uid = getuid(); //get normal uid
printf(...);
//format string error, corrupt pw->pw_uid
...
seteuid(pw->pw_uid); //use the corrupted data
```

Direct data manipulation in a vulnerable web server wu-ftpd.

DDM Example

```c
struct mystruct {
    int value;
};
void vuln_function()
{
    char buf[64];
    int result=0, length, input;
    struct mystruct * ptr;
    recv(socket, buf, input);
    ptr->value = strlen(buf);
    while (result < ptr->value) result++;
    send(socket, &result, length);
}
```

Data pointer manipulation to infer knowledge about address space layout.

DOP Attack

- Allows an attacker to perform arbitrary computations in program memory by chaining the execution of short instruction sequences (referred to as DOP gadgets)
- The execution of DOP gadgets should follow valid paths in a CFG

- Features
  - Gadgets and code reuse
  - Stitching mechanism and ordering constraint

DOP Example

```
struct server{int *cur_max, total, type;} *srv;
 int connet_limit = MAXCONN; int *size, *type;
 char buf[MAXLEN];
 Size = &buf[8]; type = &buf[12];
…
while (connet_limit--) {
  readData(sockfd, buf);      //stack bof
  if(*type == NONE ) break;
  if(*type == STREAM)
       *size = *(srv->cur_max);
   else {
      srv->type = *type;    //assignment gadget
      srv->total += *size;  //addition gadget
   } //...(code skipped)...
}
```

Vulnerable FTP server with data-oriented gadgets [1]

Round 1:
*type is corrupted to be ‘A’, neither NONE or STREAM
size is corrupted to point to srv->type (srv+0x8)

srv->type = *type; ➔ ***size = ‘A’;**

Round 2:
*type is corrupted to be ‘B’, neither NONE or STREAM
srv is corrupted to point to (srv-0x4)
srv-0x4+0x8=srv+0x4 will be srv->total
(srv->type refers to the address of srv->total)

srv->type = *type; ➔ **srv->total = ‘B’;**

Round 3:
*type is corrupted to be neither NONE or STREAM
srv is corrupted to point to (srv-0x4)+0x4
(srv->total refers to the address of srv->total)

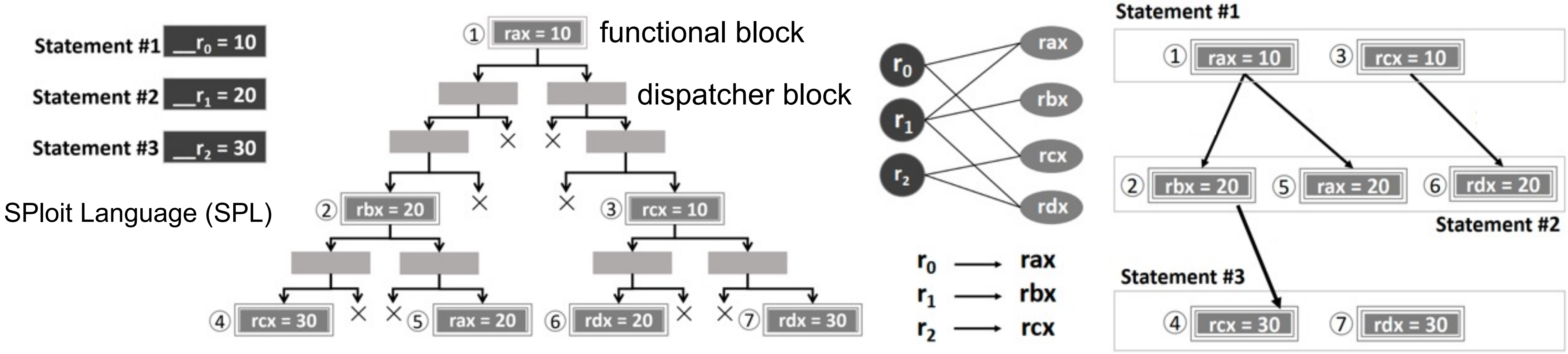
srv->total += *size; ➔ **srv->total = ‘A’ + ‘B’;**

DOP attack re-interprets gadgets for malicious purposes

[1] “Data-oriented programming: On the expressiveness of non-control data attacks,” IEEE S&P, 2016

BOP Attack

➢Unlike DOP, Block-Oriented Programming (BOP) constructs data-oriented exploits by chaining the *basic blocks* together.



Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. **Block oriented programming**: Automating data-only attacks. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 1868–1882

Challenges

- Stitching CFI-compatible gadgets is challenging
  - Require memory-write primitives to stitch gadgets
  - Involve multiple steps
    - Less evasive
  - Hard to fully automate the process of generating end-to-end DOP or BOP exploits
    - In DOP, analyze and construct exploit manually

- Defenses
  - DFI-based defenses incur high overhead of data-flow tracking

Speakers' Component in Our Tutorial Today

Daphne Yao

1. Overview of advanced attacks and various defenses.

Salman Ahmed

2. Code reuse attacks, ROP, ASLR, JITROP, and Demonstrations.

Long Cheng

3. Overview of data-oriented attacks using data manipulation.

Hans Liljestrand

4. Demonstration of DOP exploits and defenses.

N. Asokan

5. Research directions in hardware-assisted protection

Long Cheng

6. Concluding remarks and research directions.

DOP attack on ProFTPd

- Deep dive into attack by Hu *et al.* [1]
  - **You can follow along using demo docker environment: https://github.com/salmanyam/tutorial-secdev-2021**
  - **Scripts and code are in `./dop`**

- Goal to understand steps required in DOP attack
  - **Facilitates sensible security trade-offs when defending**
  - **Helps anticipate and avoid new exploitable faults in code**

[1]: Hu, *et al.* "Data-oriented programming: on the expressiveness of non-control data attacks" IEEE SP 2016

Attack steps

hands-on

- The steps of the DOP attack on ProFTPd:

- Some knowledge of memory layout (addresses and offsets)
  - **The address of `main_server` and its offset to `main_server->ServerName`**
- A dispatch loop and gadget-selector
  - **The `cmd_loop` function and the overflow in `ssreplace`**
- A set of gadgets to realize attack functionality
  - **E.g., assignment realized by exploiting `sstrncopy`**

Preventing DOP

hands-on

- How can we prevent the attack with what we now know?

1. Prevent memory errors in the first place
2. Hide information necessary for attack
3. Protect critical data from manipulation

1) Prevent: Memory safety and protection

- Can be shown using formal verification
  - **But requires considerable effort (e.g., seL4 microkernel [1])**

- Can be "improved" using run-time protection
  - **But software-based approaches often slow [2]**
  - **Typically, cannot provide full memory safety [3]**

- HW-assisted protection helps, but also increases complexity or is incomplete [4,5]

[1]: Klein, *et al.* "seL4: Formal verification of an OS kernel" ACM SIGOPS 2009
[2]: Szekeres, *et al.* "SoK: eternal war in memory," IEEE SP 2013
[3]: Gil, *et al.* "There's a hole in the bottom of the C: on the effectiveness of allocation protection" IEEE SecDev 2018
[4]: Woodruff, *et al.* "The CHERI capability model: revisiting RISC in an age of risk" ACM/IEEE ISCA 2014
[5]: Joly, *et al.* "Security analysis of CHERI ISA" Microsoft Research 2020

2) Hide: Randomization / obfuscation

- Address Space Layout Randomization (ASLR) can mitigate attacks
  - **But currently deployed implementations can be broken [1,2]**
- Re-randomization makes exploitation more challenging [3]
  - **Can have high performance impact**
- ASLR is not effective against DOP, necessarily
  - **ProFTPd demonstrates indirectly accessing data!**

- Novel hardware-assisted approaches promising
- e.g., Obfuscating all addresses and randomizing the address space [4]

[1]: Shacham, *et al.* "On the effectiveness of address-space randomization" ACM CCS 2004
[2]: Snow, *et al.* "Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization" IEEE SP 2013
[3]: Williams-King, *et al.* "Shuffler: Fast and deployable continuous code re-randomization" USENIX OSDI 2016
[4]: Gallagher, *et al.* "Morpheus: a vulnerability-tolerant secure architecture …" ACM ASPLOS 2019

3) Protect: Pointer protection

- Known attacks typically depend on data-pointer manipulation
  - **Pin-point focus on code-pointers has been successful in CFI [1]**
- Data pointers can be protected using fault-isolation [2] cryptography [3]
  - **Prevents all published DOP attacks**
- Hardware-assistance can make pointer protection faster and more secure
  - **For instance, Intel CET [4] (for code pointers) or ARM Pointer Authentication [5]**

[1]: Abadi, *et al.* "Control-flow integrity" ACM CCS 2005
[2]: Kuznetsov, *et al.* "Code-pointer integrity" USENIX OSDI 2014
[3]: Mashtizadeh, *et al.* "CCFI: Cryptographically enforced control flow integrity" ACM CCS 2015
[4]: Intel "Control-flow enforcement technology specification" 2019
[5]: Qualcomm "Pointer authentication on ARMv8.3: design and analysis of the new software security instructions" 2017

Speakers' Component in Our Tutorial Today

Daphne Yao

1. Overview of advanced attacks and various defenses.

Salman Ahmed

2. Code reuse attacks, ROP, ASLR, JITROP, and Demonstrations.

Long Cheng

3. Overview of data-oriented attacks using data manipulation.

Hans Liljestrand

4. Demonstration of DOP exploits and defenses.

N. Asokan

5. Research directions in hardware-assisted protection

Long Cheng

6. Concluding remarks and research directions.

Hardware-assisted Defenses

Protect against run-time attacks without incurring a significant performance penalty

How to thwart run-time attacks?

Run-time attacks are now routine

Software defenses incur security vs. cost tradeoffs

Hardware-assisted defenses are attractive
but deployment can be a challenge

Hardware assisted defenses in CotS processors

ARMv8-A mechanisms

- Pointer Authentication (PA)
- Memory Tagging Extension (MTE)
- Branch Target Identification (BTI)

Intel x84_64 mechanisms

- Memory Protection eXtension (MPX)
- Memory Protection Keys (PKU)
- Control-flow Enforcement Technology (CET)

ARMv8.3-A Pointer Authentication

PAuth = **PA Key management** + **PA Instructions**

General purpose hardware primitive approximating pointer integrity

- Ensure pointers in memory remain unchanged

Introduced in ARMv8.3-A specification (2016), improved in ARMv8.6-A (2020)

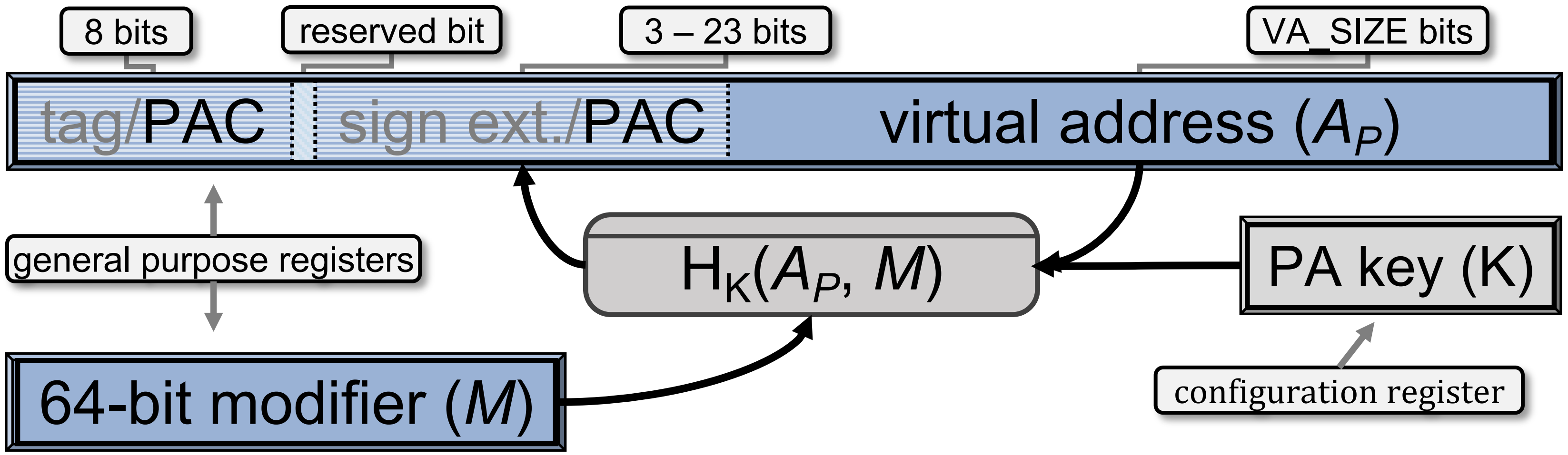
- First compatible processors 2018 (Apple A12 / iOS12)
- Userspace support in Linux 4.21, enhancements in 5.0, in-kernel support in 5.7
- Instrumentation support in GCC 7.0 (-msign-return address, deprecated in GCC 9.0, -mbranch-protection=pac-ret[+leaf] GCC 9.0 and newer)

[1]: ARM. Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile. Version E.a. July 2019
[]2: ARM. Developments in the Arm A-Profile Architecture: Armv8.6-A. September 2019

ARMv8.3-A PA – PAC Generation

Adds Pointer Authentication Code (PAC) into unused bits of pointer

- Keyed, tweakable MAC from pointer address and 64-bit modifier
- PA keys protected by hardware, modifier decided where pointer created and used

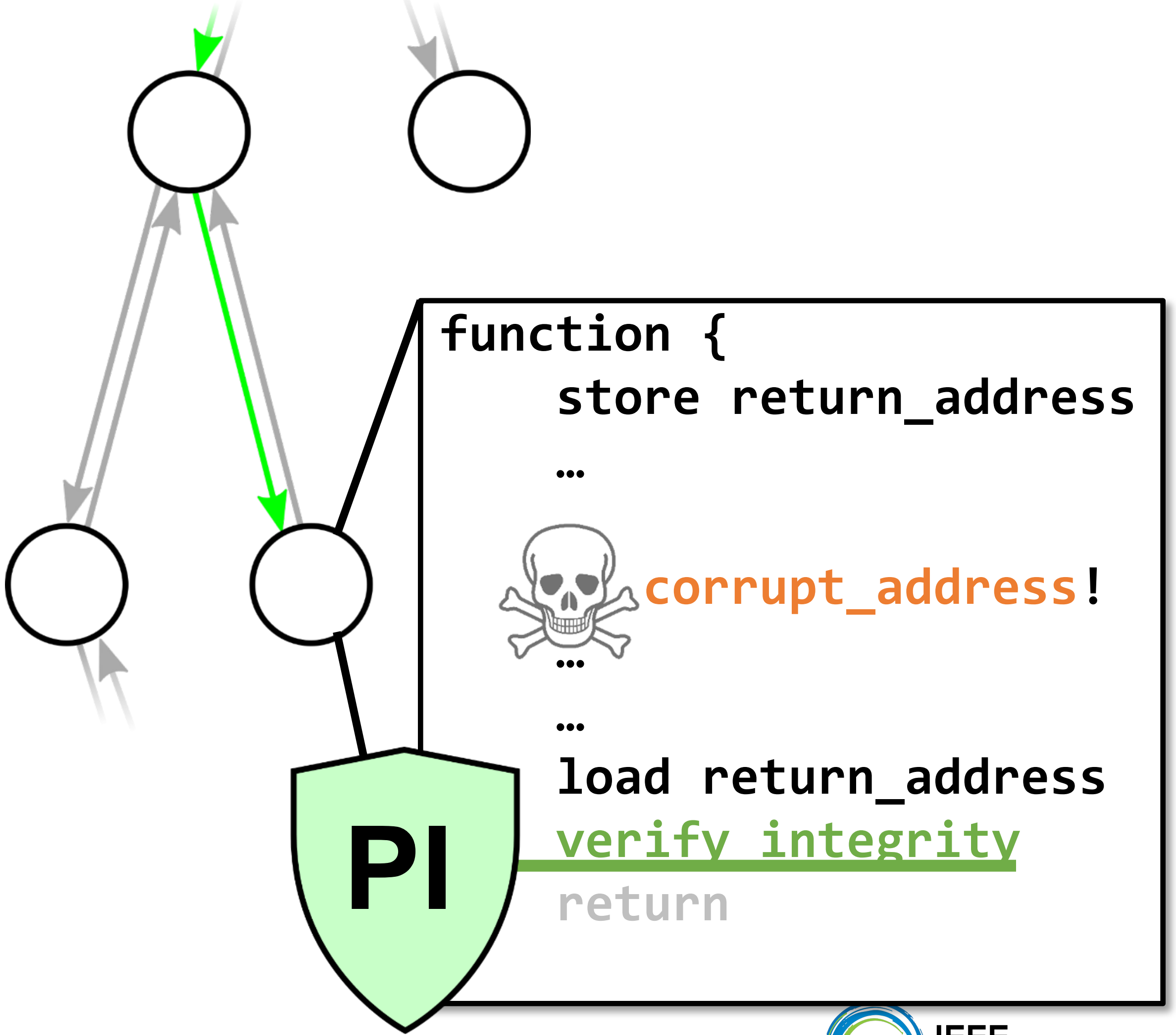


[1]: ARM. Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile. Version E.a. July 2019

Pointer Integrity: memory safety for pointers

Ensure pointers in memory remain unchanged

- Code pointer integrity implies CFI
  - Control-flow attacks manipulate code pointers
- Data pointer integrity
  - Reduces data-only attack surface



Kuznetsov *et al.* "Code-Pointer Integrity" USENIX OSDI 2014

PA-based protection schemes

PA instructions are primitives, assembled to form protection schemes

Two main components:

- When are pointers "PACed" and "unPACed"?
- Which modifier is used at a given point?

What should the modifier be for a given pointer?

- For security: using many different modifiers makes replay attacks harder
- For functionality: large numbers of modifiers are hard to keep track of

Takeaways: hardware-assisted defenses

New hardware-assisted defenses are emerging and are (going to be) widely available

How to utilize available primitives effectively?

- Towards pointer integrity with PA (USENIX SEC '19)

How to deal with downsides?

e.g. optimally minimize scope for PA reuse attacks?

- For return addresses: PACStack (USENIX SEC '21)
- For other types of pointers?

How do different hardware primitives compare?

How can we formalize run-time attacks and defenses?

https://ssg.aalto.fi/research/projects/harp/

Speakers’ Component in Our Tutorial Today

Daphne Yao

1. Overview of advanced attacks and various defenses.

Salman Ahmed

2. Code reuse attacks, ROP, ASLR, JITROP, and Demonstrations.

Long Cheng

3. Overview of data-oriented attacks using data manipulation.

Hans Liljestrand

4. Demonstration of DOP exploits and defenses.

N. Asokan

5. Research directions in hardware-assisted protection

Long Cheng

6. Concluding remarks and research directions.

Overall conclusion

- Breakdown of advanced attacks using multiple phases and factors can give us useful insights for system security assurance
  - Measuring phases/factor using metrics can quantify security parameter (e.g., re-randomization time) or attack components (e.g., gadget availability)
  - Demonstration to show various quantification methodologies with metrics
- Promises of data-oriented attacks
  - Various data-oriented attack techniques and challenges
  - Data-oriented attack demonstration
  - Data-oriented attack defenses
    - Special focus on hardware-assisted defenses
- Potential research directions

References (1)

[1]. Alexander Peslyak. “return-to-libc” attack. Bugtraq, Aug, 1997.
[2]. Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. https://users.suse.com/~krahmer/no-nx.pdf, 2005. Last accessed 10 May 2020.
[3]. Tim Newsham. Non-exec stack. Bugtraq mailing list, 2000.
[4]. Rafal Wojtczuk. The advanced return-into-lib (c) exploits: Pax case study. Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e, 2001.
[5]. Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Proceedings of the 14th ACM conference on Computer and Communications Security, pages 552–561. ACM, 2007.
[6]. Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In Proceedings of the 15th ACM conference on Computer and communications security, pages 27–38. ACM, 2008.
[7]. Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In Proceedings of the 17th ACM conference on Computer and communications security, pages 559–572. ACM, 2010.
[8]. Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, pages 30–40. ACM, 2011.
[9]. Ping Chen, Xiao Xing, Bing Mao, Li Xie, Xiaobin Shen, and Xinchun Yin. Automatic construction of jump-oriented programming shellcode (on the x86). In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, pages 20–29, 2011.
[10]. Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In 2013 IEEE Symposium on Security and Privacy, pages 574–588. IEEE, 2013.
[11]. Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In 2014 IEEE Symposium on Security and Privacy, pages 227–242. IEEE, 2014.
[12]. Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In USENIX Security Symposium, pages 385–399, 2014.
[13]. Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In 2015 IEEE Symposium on Security and Privacy, pages 745–762. IEEE, 2015.
[14]. Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. Enabling client-side crash-resistance to overcome diversification and information hiding. In NDSS, 2016.
[15]. Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, et al. Address oblivious code reuse: On the effectiveness of leakage resilient diversity. In NDSS, 2017.

References (2)

[16]. Enes Göktaş, Benjamin Kollenda, Philipp Koppe, Erik Bosman, Georgios Portokalidis, Thorsten Holz, Herbert Bos, and Cristiano Giuffrida. Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure. In 2018 IEEE European Symposium on Security and Privacy (EuroS&P), pages 227–242. IEEE, 2018.
[17]. Ali Akbar Sadeghi, Salman Niksefat, and Maryam Rostamipour. Pure-call oriented programming (pcop): chaining the gadgets using call instructions. Journal of Computer Virology and Hacking Techniques, 14(2):139–156, 2018.
[18]. PaX Team. Pax address space layout randomization (aslr). 2003.
[19]. Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual, pages 339–348. IEEE, 2006.
[20]. Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In USENIX Security Symposium, pages 475–490, 2012.
[21]. Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, and Ahmad-Reza Sadeghi. Selfrando: Securing the tor browser against de-anonymization exploits. Proceedings on Privacy Enhancing Technologies, 2016(4):454–469, 2016.
[22]. Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In Proceedings of the 2012 ACM conference on Computer and Communications Security, pages 157–168. ACM, 2012.
[23]. Yue Chen, Zhi Wang, David Whalley, and Long Lu. Remix: On-demand live randomization. In Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, pages 50–61. ACM, 2016.
[24]. Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P. Kemerlis, and Michalis Polychronakis. Compiler-assisted code randomization. In 2018 IEEE Symposium on Security and Privacy (SP), pages 461–477. IEEE, 2018.
[25]. Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. Ilr: Where'd my gadgets go? In 2012 IEEE Symposium on Security and Privacy, pages 571–585. IEEE, 2012.
[26]. William H. Hawkins, Jason D. Hiser, Michele Co, Anh Nguyen-Tuong, and Jack W. Davidson. Zipr: Efficient static binary rewriting for security. In 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pages 559–566. IEEE, 2017.
[27]. Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile-guided automated software diversity. In Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pages 1–11. IEEE Computer Society, 2013.
[28]. Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In 2015 IEEE Symposium on Security and Privacy, pages 763–780. IEEE, 2015.

References (3)

[29]. Charlie Curtsinger and Emery D. Berger. Stabilizer: Statistically sound performance evaluation. ACM SIGARCH Computer Architecture News, 41(1):219–228, 2013.
[30]. David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely rerandomization for mitigating memory disclosures. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pages 268–279. ACM, 2015.
[31]. Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. How to make aslr win the clone wars: Runtime re-randomization. In NDSS, 2016.
[32]. David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and deployable continuous code re-randomization. In OSDI, pages 367–382, 2016.
[33]. Xi Chen, Herbert Bos, and Cristiano Giuffrida. Codearmor: Virtualizing the code space to counter disclosure attacks. In 2017 IEEE European Symposium on Security and Privacy (EuroS&P), pages 514–529. IEEE, 2017.
[34]. Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürn- berger, and Jannik Pewny. 2014. You can run but you can't read: Preventing disclosure exploits in executable code. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 1342–1353.
[35]. JanWerner,GeorgeBaltas,RobDallara,NathanOtterness,KevinZ.Snow,Fabian Monrose, and Michalis Polychronakis. 2016. No-execute-after-read: Preventing code disclosure in commodity software. In Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security. ACM, 35–46.
[36]. Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2015. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In Proceed- ings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 256–267.
[37]. Michael Backes and Stefan Nürnberger. 2014. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In USENIX Security Symposium. 433–447.
[38]. Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. 2015. ASLR-Guard: Stopping address space leakage for code reuse attacks. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 280–291.
[39]. Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In 2016 IEEE Symposium on Security and Privacy (SP). IEEE, 969–986.
[40]. Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, 1868–1882.

References (4)

[41]. Salman Ahmed, Ya Xiao, Kevin Z. Snow, Gang Tan, Fabian Monrose, and Danfeng (Daphne) Yao. 2020. Methodologies for Quantifying (Re-)randomization Security and Timing under JIT-ROP. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), 1803–1820. DOI:https://doi.org/10.1145/3372297.3417248

[42]. Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Returnoriented programming: Systems, languages, and applications. ACM Transactionson Information and System Security (TISSEC) 15, 1 (2012), 2