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
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**

\[
\begin{align*}
\text{Assessing impact of defenses on attack components.} \\
\text{choosing effective security parameters.} \\
\text{Improving awareness on system security.}
\end{align*}
\]

Necessary for system security assurance
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\textsuperscript{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

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

Phishing → Vulnerability → Copying data → Monitoring traffic

Reconnaissance → Gateway → Execution → Command & Control

Identification → Construction → Entrance → Exploit payload

Target, gateways, etc. → Payload, connectivity → Input control → root shell, malware installation, etc.

Reuse code from binaries

Buffer overflow, use-after-free, etc.
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

1997-2005
Return-to-libc [1, 2, 3, 4]

2007-2010
ROP [5, 6, 7]

2011
JOP [8, 9]

2013
JITROP [10]

2014
BROP [11]

2015
COP [12]

2016
COOP [13]

2017
CROP [14]

2018
AOCR [15]

PIROP [16]

PCOP [17]

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)

- Coarse-grained randomization: PaX ASLR’03 [18]
- Function-level randomization: ASLP’06 [19], ASR’12 [20], Selfrando’16 [21]
- Block-level randomization: Binary Stirring’12 [22], Remix’16 [23], CCR’18 [24]
- Instruction-level randomization: ILR’12 [25], Zipr’17 [26]
- Register-level randomization: MCR’13 [27], Readactor’15 [28]

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

<table>
<thead>
<tr>
<th>Application address space</th>
<th>Application address space</th>
<th>Application address space</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08040600 .TEXT</td>
<td>0x08040600 .TEXT</td>
<td>0x08040600 Heap</td>
</tr>
<tr>
<td>0x08040700 Heap</td>
<td>0x08040700 Heap</td>
<td>0x08040700 .DATA</td>
</tr>
<tr>
<td>0x08040800 Stack</td>
<td>0x08040800 Stack</td>
<td>0x08040800 .TEXT</td>
</tr>
<tr>
<td>0x08040900 .GOT</td>
<td>0x08040900 .GOT</td>
<td>0x08040900 .DATA</td>
</tr>
<tr>
<td>0x08040a00 .DATA</td>
<td>0x08040a00 .DATA</td>
<td>0x08040a00 .GOT</td>
</tr>
<tr>
<td>0x08040b00 Library</td>
<td>0x08040b00 Library</td>
<td>0x08040b00 Stack</td>
</tr>
</tbody>
</table>

Run 1 Run 2 Run 3
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

1. Overview of advanced attacks and various defenses.
   - Daphne Yao

2. Code reuse attacks, ROP, ASLR, JITROP, and Demonstrations.
   - Salman Ahmed

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

4. Demonstration of DOP exploits and defenses.
   - Hans Liljestrand

5. Research directions in hardware-assisted protection
   - N. Asokan

6. Concluding remarks and research directions.
   - Long Cheng
- Code Reuse Attacks,
- Return-Oriented Programming (ROP),
- Just-In Time ROP (JITROP), and
- Demonstration
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.

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

<table>
<thead>
<tr>
<th>Whole functions</th>
<th>Gadgets</th>
</tr>
</thead>
<tbody>
<tr>
<td>system()</td>
<td>0x0808A24: pop eax; ret</td>
</tr>
<tr>
<td>exit()</td>
<td>0x0808C46: pop ebx; ret</td>
</tr>
<tr>
<td></td>
<td>0x0808A058: mov [ebx], eax; ret</td>
</tr>
</tbody>
</table>
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]

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

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

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

<table>
<thead>
<tr>
<th>Technique</th>
<th>ROP gadgets</th>
<th>System functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Stack pointer (SP) manipulation</td>
<td>XCHG EAX, EDX</td>
<td>VirtualProtect</td>
</tr>
<tr>
<td>- Manipulation through charstring program</td>
<td>MOV EBX, EDX</td>
<td>GetProcAddress()</td>
</tr>
<tr>
<td>- ROP gadgets</td>
<td>POP ESI</td>
<td>LoadLibrary()</td>
</tr>
<tr>
<td>- System functions</td>
<td>POP ECX</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REP MOVSD</td>
<td>NtGdiAddRemoteFontToDC</td>
</tr>
<tr>
<td></td>
<td>JMP EBX</td>
<td></td>
</tr>
</tbody>
</table>
- **Coarse-grained ASLR**
  
  **Key Limitation**: Can be bypassed using information leaks

- **Fine-grained ASLR**
  
  **Goal**: Aims to protect information leaks
Does then fine-grained ASLR make code reuse attacks impossible?

No!!!
Just-In-Time Return-Oriented Programming\textsuperscript{2} (JITROP) [10]

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

$P_1, P_2, \ldots, P_n$ are 4 KB code pages
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]

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].

<table>
<thead>
<tr>
<th>Gadget set</th>
<th>Minimum (s)</th>
<th>Average (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC</td>
<td>2.2</td>
<td>4.3</td>
</tr>
<tr>
<td>Priority</td>
<td>1.5</td>
<td>3.5</td>
</tr>
<tr>
<td>MOV TC</td>
<td>3.5</td>
<td>5.3</td>
</tr>
<tr>
<td>Payload*</td>
<td>2.1</td>
<td>4.8</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>2.3s</strong></td>
<td><strong>4.5s</strong></td>
</tr>
</tbody>
</table>

* 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.

<table>
<thead>
<tr>
<th>Randomization schemes</th>
<th>Granularity</th>
<th>↓ (%) MIN-FP</th>
<th>↓ (%) EX-FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main executables</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst. level rando. [50]</td>
<td>Inst.</td>
<td>79.7</td>
<td>82.5</td>
</tr>
<tr>
<td>Func. level rando. [25]</td>
<td>FB</td>
<td>27.63</td>
<td>36.55</td>
</tr>
<tr>
<td>Func.+Reg. level rando. [53]</td>
<td>FB &amp; Reg.</td>
<td>17.62</td>
<td>42.37</td>
</tr>
<tr>
<td>Block level rand. [59]</td>
<td>BB</td>
<td>19.58</td>
<td>44.64</td>
</tr>
<tr>
<td>Dynamic libraries</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst. level rando. [50]</td>
<td>Inst.</td>
<td>81.3</td>
<td>92.2</td>
</tr>
<tr>
<td>Func. level rando. [25]</td>
<td>FB</td>
<td>46.5</td>
<td>43.8</td>
</tr>
<tr>
<td>Func.+Reg. level rando. [53]</td>
<td>FB &amp; Reg.</td>
<td>44.2</td>
<td>43.9</td>
</tr>
<tr>
<td>Block level rand. [59]</td>
<td>BB</td>
<td>20.98</td>
<td>37.0</td>
</tr>
</tbody>
</table>

Reduction of Turing-complete gadget set with different randomization schemes
Impact of the Location of Pointer Leakage

No impact on connectivity

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 privileged 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 outputs 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

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

0x08041800
0x0804B800
0x08045800

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

i) gadget availability and
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

1. Overview of advanced attacks and various defenses.
   - Daphne Yao

2. Code reuse attacks, ROP, ASLR, JITROP, and Demonstrations.
   - Salman Ahmed

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

4. Demonstration of DOP exploits and defenses.
   - Hans Liljestrand

5. Research directions in hardware-assisted protection
   - N. Asokan

6. Concluding remarks and research directions.
   - Long Cheng
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**: Data-Oriented Programming [Hu et al., S&P]
- **2017**: Undermine CFI protection [Davi. et al., NDSS]
- **2018**: Block-Oriented Programming [Ispoglou et al., CCS]
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

```c
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

Vulnerable FTP server with data-oriented gadgets[1]

1. `struct server{int *cur_max, total, type;} *srv;`
2. `int connet_limit = MAXCONN; int *size, *type;`
3. `char buf[MAXLEN];`
4. `Size = &buf[8]; type = &buf[12];`
5. `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)...
} `

DOP attack re-interprets gadgets for malicious purposes

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;  \rightarrow \hspace{1em} \ast 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;  \rightarrow \hspace{1em} \ast 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;  \rightarrow \hspace{1em} \ast srv->total = ‘A’ + ‘B’;`

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

1. Overview of advanced attacks and various defenses.
   - Daphne Yao

2. Code reuse attacks, ROP, ASLR, JITROP, and Demonstrations.
   - Salman Ahmed

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

4. Demonstration of DOP exploits and defenses.
   - Hans Liljestrand

5. Research directions in hardware-assisted protection
   - N. Asokan

6. Concluding remarks and research directions.
   - Long Cheng
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

Attack steps

• 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

• 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]

[3]: Gil, et al. “There’s a hole in the bottom of the C: on the effectiveness of allocation protection” IEEE SecDev 2018
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 \[^{4}\]

---

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
[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

1. Overview of advanced attacks and various defenses.
   - Daphne Yao

2. Code reuse attacks, ROP, ASLR, JITROP, and Demonstrations.
   - Salman Ahmed

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

4. Demonstration of DOP exploits and defenses.
   - Hans Liljestrand

5. Research directions in hardware-assisted protection
   - N. Asokan

6. Concluding remarks and research directions.
   - Long Cheng
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 x86_64 mechanisms
- Memory Protection eXtension (MPX)
- Memory Protection Keys (PKU)
- Control-flow Enforcement Technology (CET)
ARMv8.3-A Pointer Authentication

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)

[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

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
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](https://ssg.aalto.fi/research/projects/harp/))

How to deal with downsides?
e.g. optimally minimize scope for PA reuse attacks?
- For return addresses: PACStack ([USENIX SEC ’21](https://ssg.aalto.fi/research/projects/harp/))
- For other types of pointers?

How do different hardware primitives compare?

How can we formalize run-time attacks and defenses?
Speakers’ Component in Our Tutorial Today

1. Overview of advanced attacks and various defenses.
   - Daphne Yao

2. Code reuse attacks, ROP, ASLR, JITROP, and Demonstrations.
   - Salman Ahmed

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

4. Demonstration of DOP exploits and defenses.
   - Hans Liljestrand

5. Research directions in hardware-assisted protection
   - N. Asokan

6. Concluding remarks and research directions.
   - Long Cheng
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)

[4]. Rafal Wojtczuk. The advanced return-into-lib (c) exploits: Pax case study. Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile # 0x04 of 0x0e, 2001.
References (2)


References (3)

References (4)