Conceal ROP gadgets for AArch64 COTS binary

Dongli Zhang

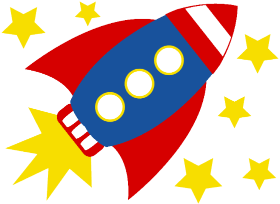
Oracle Asia Research and Development Centers (Beijing)

dongli.zhang@oracle.com

December 11, 2017

Plan

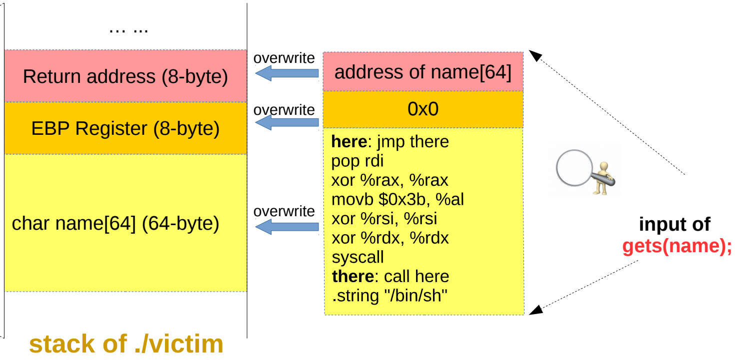
- ROP Attack: Return Oriented Programming Attack
- ELF and AArch64
- NORAX: eXecute-Only-Memory (XOM) on AArch64



Code Injection Attack

- Stack Smashing: to inject and run shellcode in stack
- Linux x86_64 Calling Convention: RDI, RSI, RDX, RCX, R8, R9, XMM07

```
/* victim.c */
int main()
{
    char name[64];
    puts("What's your name?");
    gets(name);
    printf("Hello, %s!\n", name);
    return 0;
}
```



Stack Smashing Mitigations

- Stack Canary
  - StackGuard: Automatic Adaptive Detectionand Prevention of Buffer-Overflow Attacks. USENIX Security 1998.

Stack Smashing Mitigations

- Stack Canary
  - StackGuard: Automatic Adaptive Detectionand Prevention of Buffer-Overflow Attacks. USENIX Security 1998.
  - To disable via: *gcc -fno-stack-protector -o victim victim.c*

Stack Smashing Mitigations

- Stack Canary
  - StackGuard: Automatic Adaptive Detectionand Prevention of Buffer-Overflow Attacks. USENIX Security 1998.
  - To disable via: *gcc -fno-stack-protector -o victim victim.c*
- DEP (Data Execution Prevention): W$\oplus$X (NX bit)
  - Attackers are not able to execute any injected code!
  - To disable via: *execstack -s victim*

Stack Smashing Mitigations

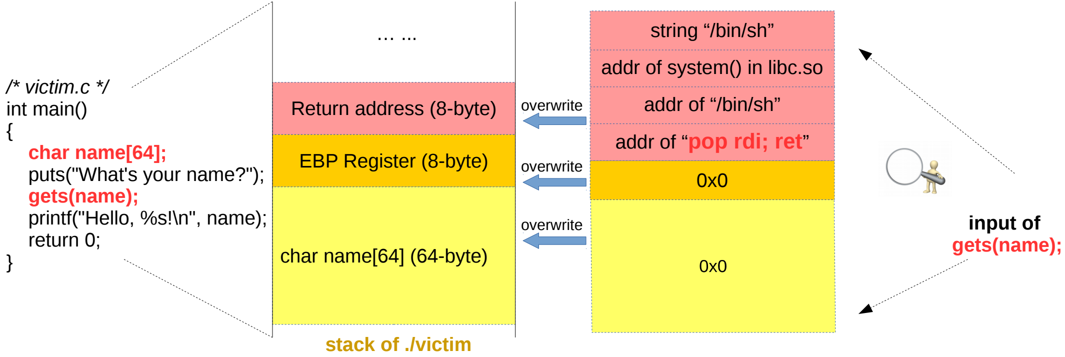
- Stack Canary
  - StackGuard: Automatic Adaptive Detectionand Prevention of Buffer-Overflow Attacks. USENIX Security 1998.
  - To disable via: *gcc -fno-stack-protector -o victim victim.c*
- DEP (Data Execution Prevention): W$\oplus$X (NX bit)
  - Attackers are not able to execute any injected code!
  - To disable via: *execstack -s victim*
- ASLR (Address Space Layout Randomization)
  - Transparent runtime randomization for security. SRDS 2003.

Stack Smashing Mitigations

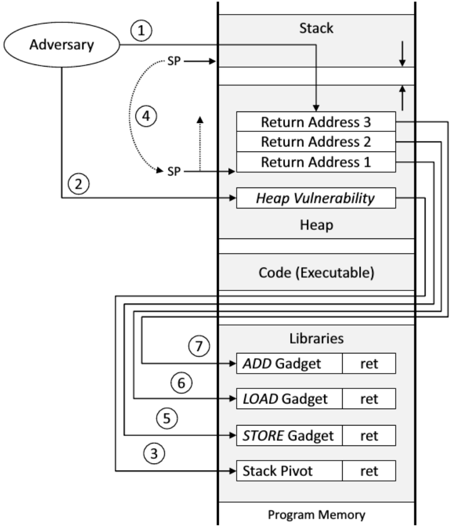
- Stack Canary
  - StackGuard: Automatic Adaptive Detectionand Prevention of Buffer-Overflow Attacks. USENIX Security 1998.
  - To disable via: *gcc -fno-stack-protector -o victim victim.c*
- DEP (Data Execution Prevention): W$\oplus$X (NX bit)
  - Attackers are not able to execute any injected code!
  - To disable via: *execstack -s victim*
- ASLR (Address Space Layout Randomization)
  - Transparent runtime randomization for security. SRDS 2003.
  - To disable via: *setarch ‘arch‘ -R ./victim*
  - To disable via: *echo 0 > /proc/sys/kernel/randomize_va_space*

Code Reuse Attack (1/2)

- Gadgets: instruction sequence ended with "ret" instruction within existing program or libraries already present in memory
- ROP (Return Oriented Programming): to perform arbitrary operations by chaining relavant gadgets to bypass DEP

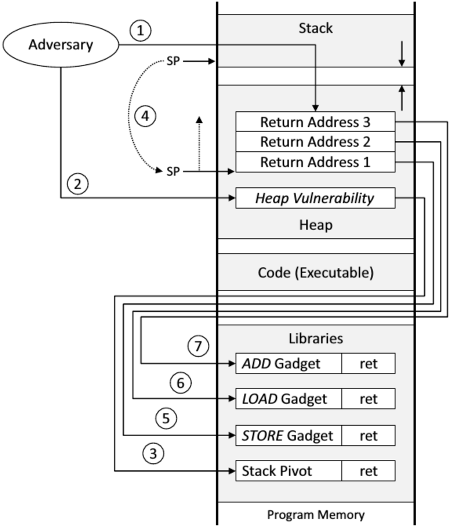


Code Reuse Attack (2/2)



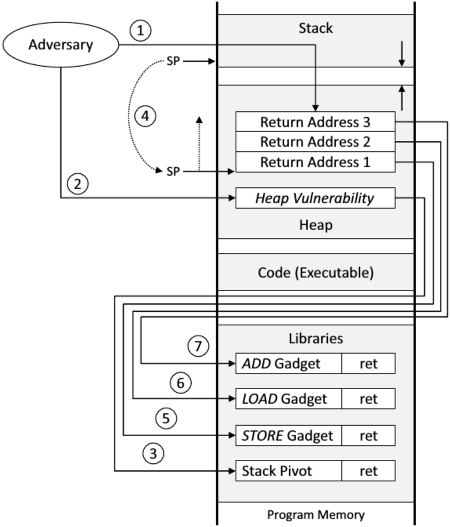
1. inject ROP payload

Code Reuse Attack (2/2)



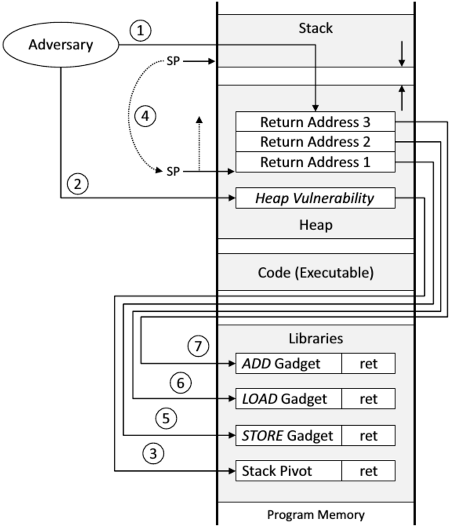
1. inject ROP payload
2. hijack control flow

Code Reuse Attack (2/2)



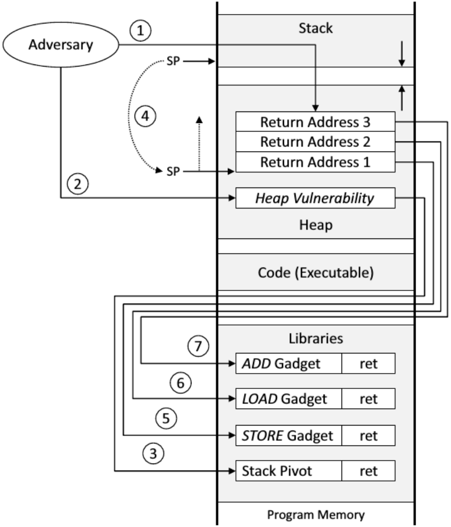
1. inject ROP payload
2. hijack control flow
3. stack pivot sequences (e.g., mov %eax, %esp; ret)

Code Reuse Attack (2/2)



1. inject ROP payload
2. hijack control flow
3. stack pivot sequences (e.g., mov %eax, %esp; ret)
4. "ret" redirects to ROP payload

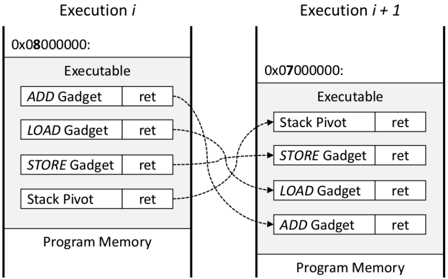
Code Reuse Attack (2/2)



1. inject ROP payload
2. hijack control flow
3. stack pivot sequences (e.g., mov %eax, %esp; ret)
4. "ret" redirects to ROP payload
5. ROP gadget and ret
6. ROP gadget and ret
7. ROP gadget and ret

Fine-Grained Address Space Layout Randomization (ASLR)

- function order permutation
- basic block order permutation
- swap registers and replace instructions
- instruction location randomization

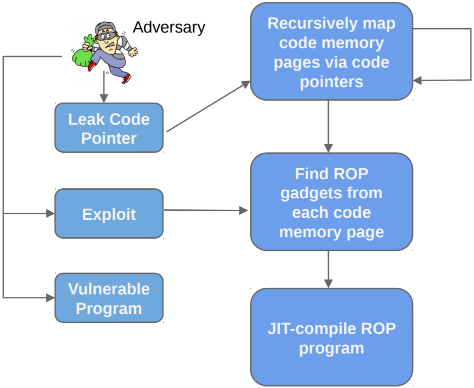


Just-In-Time Return Oriented Programming Attack

- Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. IEEE S&P (Oakland) 2013

Thread Model Assumption

- Exercise a vulnerable entry point
- Execute arbitrary malicious computations



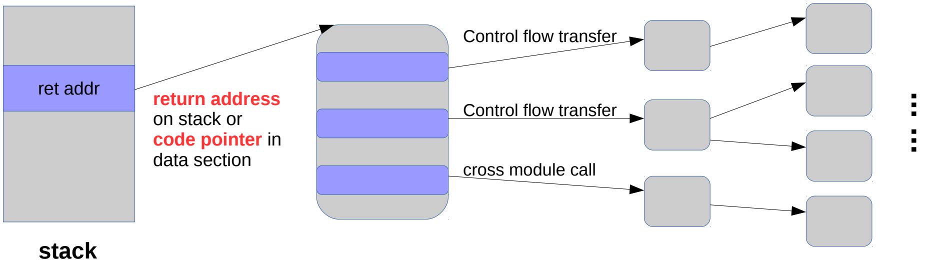
Memory Disclosure

Direct Memory Disclosure

- read instructions in code page

Indirect Memory Disclosure

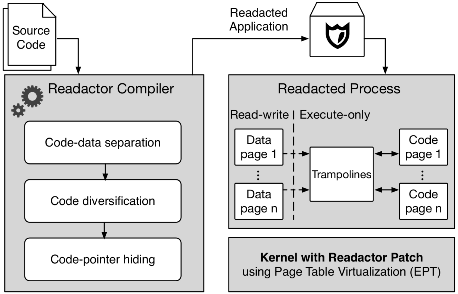
- return address
- function pointer
- dynamic linking information
- c++ vtable & exception handler



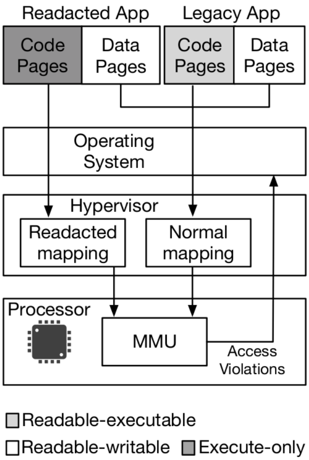
Readactor 1/2

Readactor: Practical Code Randomization Resilient to Memory Disclosure. IEEE S & P 2015

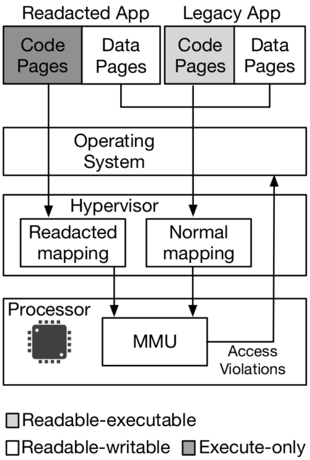
- Fine-grained code diversification via LLVM
- Code and data separation via Intel EPT and LLVM
- Code-pointer hiding via LLVM
- Does not support COTS binary

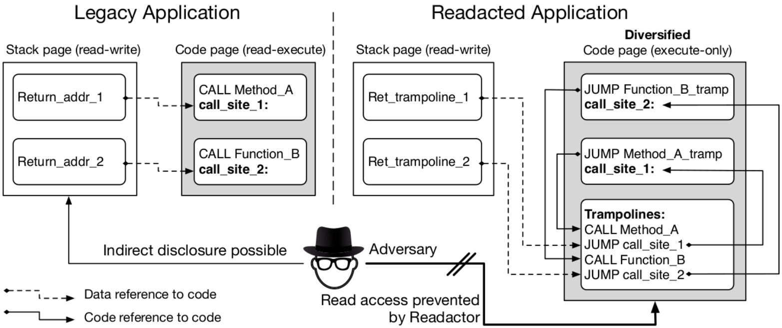


Readactor 2/2



Readactor 2/2





New and Hard Problem

- Enable XOM on Android AArch64 COTS binaries (NORAX)
- ~~Hide code pointers in data section (future work)~~

COTS Binary - Commercial Off-The-Shelf

- # aarch64-linux-gnu-strip <binary>
- without symbol information

```
0000000000002460 <main>:
   2460:  d111c3ff    sub sp, sp, #0x470
   2464:  a9ba7bfd    stp x29, x30, [sp,#-96]!
   2468:  910003fd    mov x29, sp
   … …
0000000000003484 <_start>:
   3484:  8b3f63e0    add x0, sp, xzr
   3488:  17ffffea    b  3430 <do_arm64_start>
000000000000348c <__atexit_handler_wrapper>:
   348c:  a9be7bfd    stp x29, x30, [sp,#-32]!
   3490:  910003fd    mov x29, sp
   … …
00000000000034b4 <atexit>:
   34b4:  a9be7bfd    stp x29, x30, [sp,#-32]!
   34b8:  910003fd    mov x29, sp
```

Original Binary

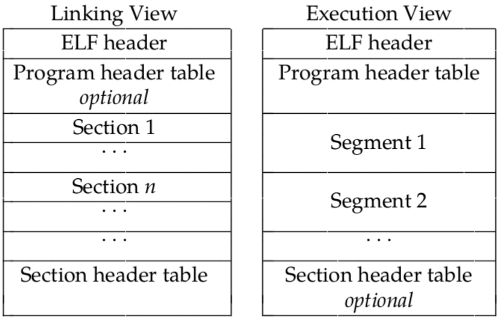
aarch64-linux-gnu-strip

```
2460:  d111c3ff    sub sp, sp, #0x470
2464:  a9ba7bfd    stp x29, x30, [sp,#-96]!
2468:  910003fd    mov x29, sp
  … …
3484:  8b3f63e0    add x0, sp, xzr
3488:  17ffffea    b  3430
348c:  a9be7bfd    stp x29, x30, [sp,#-32]!
3490:  910003fd    mov x29, sp
  … …
34b4:  a9be7bfd    stp x29, x30, [sp,#-32]!
34b8:  910003fd    mov x29, sp
```

COTS Binary

ELF - Linking vs. Execution

- segments sample
  - INTERP
  - LOAD
  - DYNAMIC
- sections sample
  - .interp
  - .dynsym, .dynamic
  - .rela.dyn, .rela.plt, .got.plt, .got
  - .plt, .text
  - .data, .rodata, .bss
- manuals
  - Executable and Linkable Format (ELF)
  - ELF for the ARM Architecture
  - ELF for the ARM 64-bit Architecture (AArch64)



ELF - load executable ELF

1. user space loads executable binary via exec system call

ELF - load executable ELF

1. user space loads executable binary via exec system call
2. kernel loads executable binary and dynamic linker into memory

ELF - load executable ELF

1. user space loads executable binary via exec system call
2. kernel loads executable binary and dynamic linker into memory
3. dynamic linker performs linking jobs while loading all prerequisite libraries (android is without lazy address resolution)

ELF - load executable ELF

1. user space loads executable binary via exec system call
2. kernel loads executable binary and dynamic linker into memory
3. dynamic linker performs linking jobs while loading all prerequisite libraries (android is without lazy address resolution)
4. start the executable binary

ELF - load executable ELF

1. user space loads executable binary via exec system call
2. kernel loads executable binary and dynamic linker into memory
3. dynamic linker performs linking jobs while loading all prerequisite libraries (android is without lazy address resolution)
4. start the executable binary
5. resolve dynamic symbol on-demand by linker

ELF - resolve dynamic symbol

Suppose ./test calls function puts() belong to libc (lazy address resolution):

1. ./test calls puts@plt belong to plt section

ELF - resolve dynamic symbol

Suppose ./test calls function puts() belong to libc (lazy address resolution):

1. ./test calls puts@plt belong to plt section
2. puts@plt redirects to puts in got.plt which points to corresponding handler in ld

ELF - resolve dynamic symbol

Suppose ./test calls function puts() belong to libc (lazy address resolution):

1. ./test calls puts@plt belong to plt section
2. puts@plt redirects to puts in got.plt which points to corresponding handler in ld
3. ld calculates the hash of symbol name (puts), traverses each libraries and searches in buckets of gnu.hash with the hash value to identify the index of puts() in dynsym section

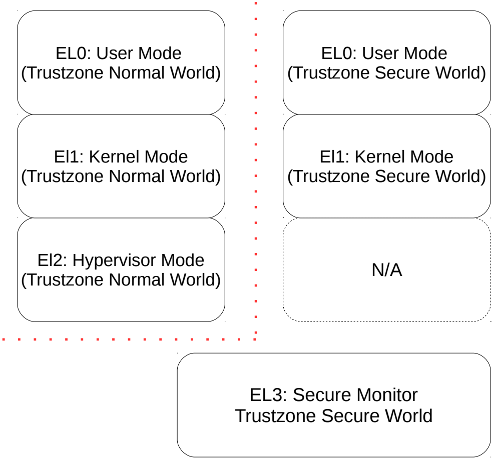
ELF - resolve dynamic symbol

Suppose ./test calls function puts() belong to libc (lazy address resolution):

1. ./test calls puts@plt belong to plt section
2. puts@plt redirects to puts in got.plt which points to corresponding handler in ld
3. ld calculates the hash of symbol name (puts), traverses each libraries and searches in buckets of gnu.hash with the hash value to identify the index of puts() in dynsym section
4. Once entry of puts in dynsym is identified, the address of puts would be written to got.plt with the help of binary's rela.plt

AArch64

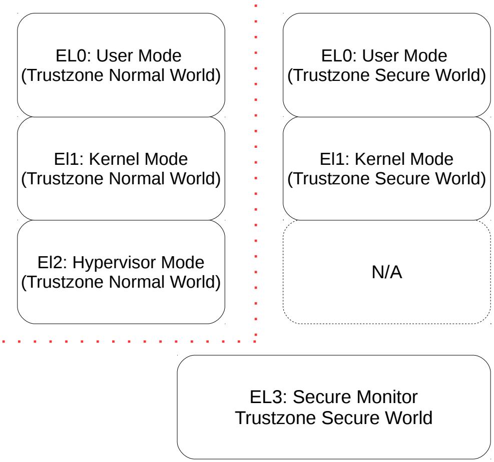
- instructions: 4-byte aligned and fixed size



AArch64 CPU Exception Level

AArch64

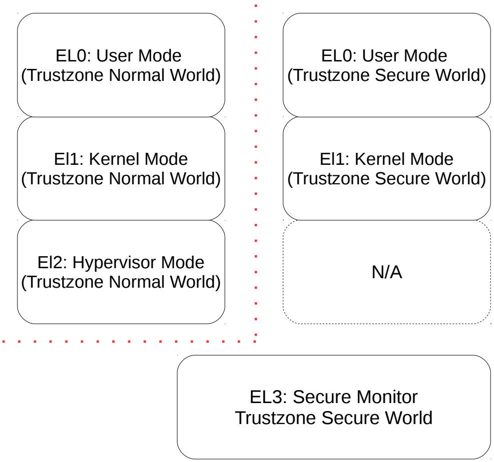
- instructions: 4-byte aligned and fixed size
- mode: user (EL0), kernel (EL1), hypervisor (EL2) and secure monitor (EL3)



AArch64 CPU Exception Level

AArch64

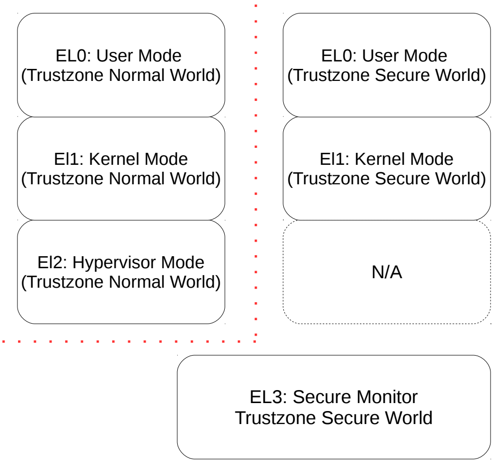
- instructions: 4-byte aligned and fixed size
- mode: user (EL0), kernel (EL1), hypervisor (EL2) and secure monitor (EL3)
- registers: X0-X30 (X29 is FP, X30 is LR), SP (PC is not accessible)



AArch64 CPU Exception Level

AArch64

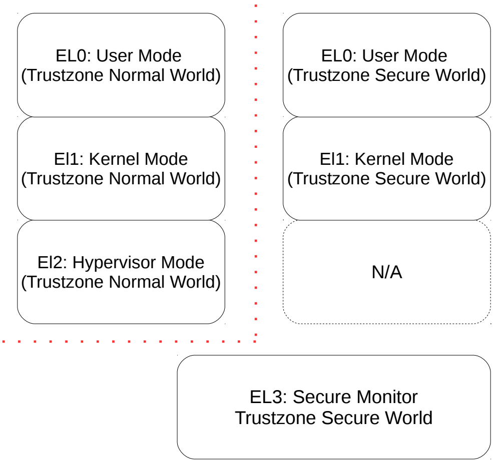
- instructions: 4-byte aligned and fixed size
- mode: user (EL0), kernel (EL1), hypervisor (EL2) and secure monitor (EL3)
- registers: X0-X30 (X29 is FP, X30 is LR), SP (PC is not accessible)
- branch: B, BL (Branch with Link), BLR (Branch with Link to Register), BR (Branch to Register), B.cond



AArch64 CPU Exception Level

AArch64

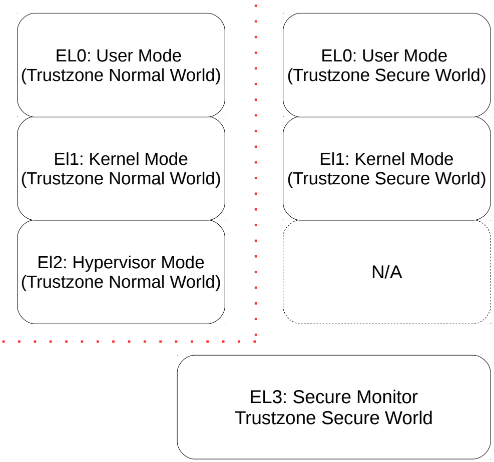
- instructions: 4-byte aligned and fixed size
- mode: user (EL0), kernel (EL1), hypervisor (EL2) and secure monitor (EL3)
- registers: X0-X30 (X29 is FP, X30 is LR), SP (PC is not accessible)
- branch: B, BL (Branch with Link), BLR (Branch with Link to Register), BR (Branch to Register), B.cond
- memory load: ADR, ADRP, LDR



AArch64 CPU Exception Level

AArch64

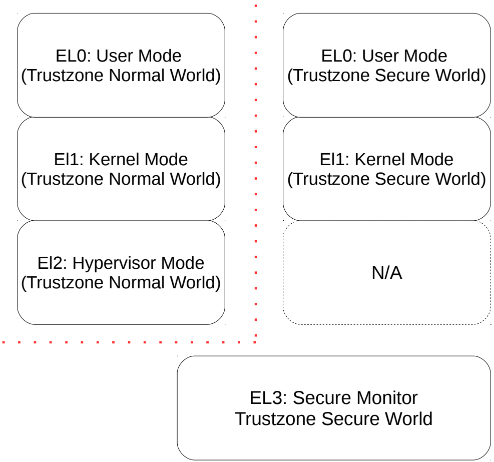
- instructions: 4-byte aligned and fixed size
- mode: user (EL0), kernel (EL1), hypervisor (EL2) and secure monitor (EL3)
- registers: X0-X30 (X29 is FP, X30 is LR), SP (PC is not accessible)
- branch: B, BL (Branch with Link), BLR (Branch with Link to Register), BR (Branch to Register), B.cond
- memory load: ADR, ADRP, LDR
- other: ARM Architecture Reference Manual: ARMv8, for ARMv8-A architecture profile



AArch64 CPU Exception Level

AArch64

- instructions: 4-byte aligned and fixed size
- mode: user (EL0), kernel (EL1), hypervisor (EL2) and secure monitor (EL3)
- registers: X0-X30 (X29 is FP, X30 is LR), SP (PC is not accessible)
- branch: B, BL (Branch with Link), BLR (Branch with Link to Register), BR (Branch to Register), B.cond
- memory load: ADR, ADRP, LDR
- other: ARM Architecture Reference Manual: ARMv8, for ARMv8-A architecture profile
- since Android 5.0 (Lolopop), non-PIE loading is no longer supported



AArch64 CPU Exception Level

NORAX: Enabling Execute-Only Memory for COTS Binaries on AArch64

[*] YaohuiChen,[*] DongliZhang,[†] RuowenWang,[*] RuiQiao,
[†] AhmedM.Azab,[*] LongLu,[†] HayawardhVijayakumar,[†] WenboShen

[*]Stony Brook University [†]Samsung Research America

IEEE Symposium on Security & Privacy (Oakland) 2017

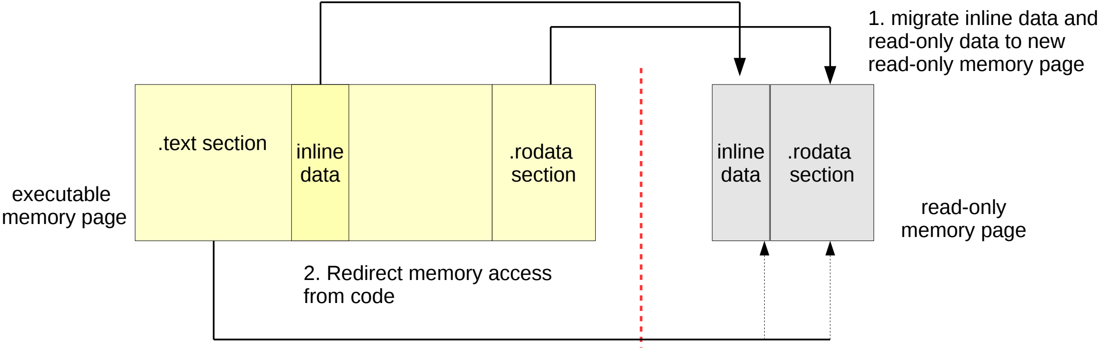
XOM on AArch64

- commit, revert and commit
  1. 2016-08-25, arm64: Introduce execute-only page access permissions
  2. 2014-05-16, Revert "arm64: Introduce execute-only page access permissions"
  3. 2014-05-09, arm64: Introduce execute-only page access permissions
- last commit (2016-08-25): cab15ce604e550020bb7115b779013b91bcdbc21
- gcc/llvm (AFAIK) does not support code-data seperation

AP[2:1]	EL0 Permission	EL1 Permission
00	Executable-only	Read/Write
01	Read/Write, Config-Executable	Read/Write
10	Executable-only	Read-only
11	Read, Executable	Read-only

NORAX Solution

1. separate data and code to different pages
2. properly update all references



Executable Data Relocation Sample

```
3e34:   90000032   adrp   x18, 7000        (struck)
3e34:   90000032   adrp   x18, 14000
3e38:   91104240   add x0, x18, #0x410     (struck)
3e38:   91104240   add x0, x18, #0x440
3e3c:   97fff8cd   bl  2170 <puts@plt>
… …
… …
… …
6fd0: rodata                               (struck)
… …
… …
14000: new rodata
```

Read-only Data Relocation

```
5190:   5c001341   ldr d1, 53f8            (struck)
5190:   xxxxxxxx   b 7000
5194:   1e612040   fcmp   d2, d1
… …
53f8:   ffffffff   .inst   0xffffffff
53fc:   7fefffff   .inst   0x7fefffff
5400:   52d0e560   .inst   0x52d0e560
… …
7000:   xxxxxxxx   ldr d1,  143f8
7004:   xxxxxxxx   b 5194
… …
143f8:   ffffffff   .inst   0xffffffff
143fc:   7fefffff   .inst   0x7fefffff
14400:   52d0e560   .inst   0x52d0e560
```

duplicate inline data

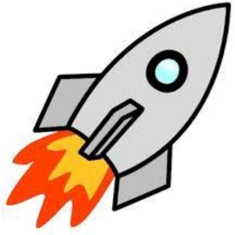
Inline Data Relocation

NORAX Challenges

- rodata and executable inline data
  - Reference from code (.text)
  - Reference from symbol table (.dynsym)
  - Reference from relocation table (.rela.dyn)
  - Reference from global offset table (.got)
  - Reference from read-only global data (.data.rel.ro)

NORAX Challenges

- rodata and executable inline data
  - Reference from code (.text)
  - Reference from symbol table (.dynsym)
  - Reference from relocation table (.rela.dyn)
  - Reference from global offset table (.got)
  - Reference from read-only global data (.data.rel.ro)
- read-only ELF header
  - Reference from linker



NORAX Challenges

- rodata and executable inline data
  - Reference from code (.text)
  - Reference from symbol table (.dynsym)
  - Reference from relocation table (.rela.dyn)
  - Reference from global offset table (.got)
  - Reference from read-only global data (.data.rel.ro)
- read-only ELF header
  - Reference from linker
- .eh_frame_hdr/.eh_frame
  - Reference from C++ runtime

Design Goals

Code-Data Separation: precision vs. practical

- A complete set of executable data
- A subset of references

Design Goals

Code-Data Separation: precision vs. practical

- A complete set of executable data
- A subset of references

Security

- Expose as less code as possible
- Enforce policy based security on missed references

Design Goals

Code-Data Separation: precision vs. practical

- A complete set of executable data
- A subset of references

Security

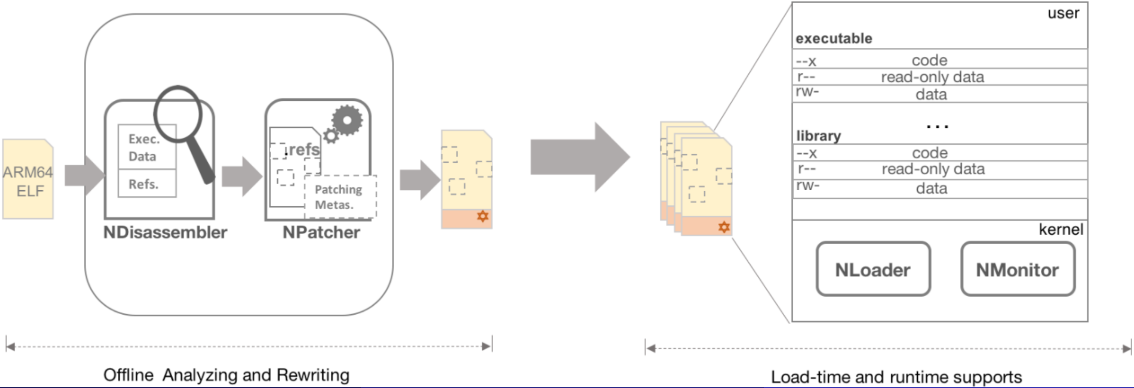
- Expose as less code as possible
- Enforce policy based security on missed references

Practicability

- Low runtime and memory overhead
- Non-exclusive binary hardening solution
- Backward compatibility
- Modularity support

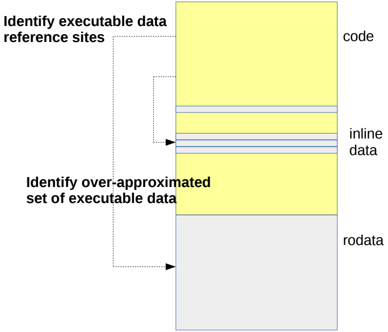
NORAX Framework

- NDisassembler: collect executable data and references
- NPatcher: static binary transformation
- NLoader: update executable data references
- NMonitor: runtime policy check for false-positive



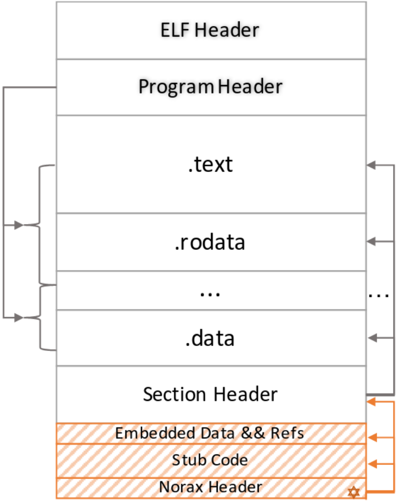
NORAX: NDisassembler

- *Algorithm 1* and *Algorithm 2* in NORAX paper for details
  1. Linear-sweep disassembly (objdump -d)
  2. Identify executable data position (rodata or inline) and reference (adr(p) or ldr)
  3. For unbounded data, collect a set of over-approximated date via Unbounded Data Expansion (Algorithm 2)



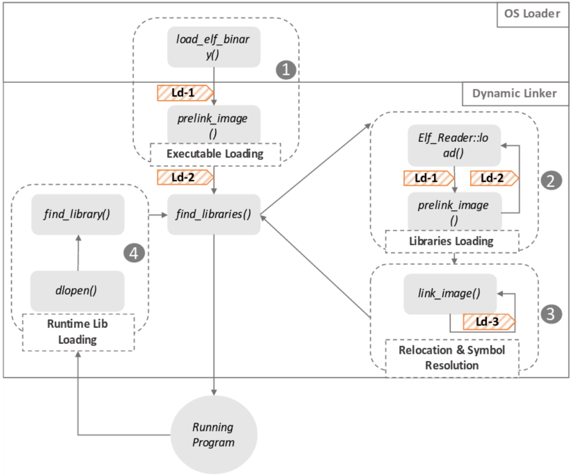
NORAX: NPatcher

- New memory layout
  - New location of the executable data
  - Take into consideration reference addressing range, and emit stub code if needed
- Append NORAX-related metadata to the end
  - Duplicated inline data
  - References locations and displacements
  - Stub code
  - NORAX header



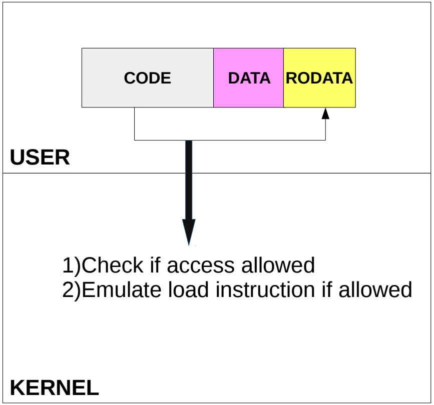
NORAX: NLoader

- **Ld-1:** Setup NORAX book-keeping data and new mapping of read-only data and sections
- **Ld-2:** Redirect .dynamic access to new read-only sections
- **Ld-3:** Adjust all referencees and enable XOM



NORAX: NMonitor

- Missed reference to embedded data
  - NDisassembler may miss some references
- Reference to .eh_frame_hdr and .eh_frame



Evaluation - transformation correctness

- LG Nexus 5X (Qualcomm Snapdragon 808MSM8992 (4 x ARM Cortex-A53 & 2 x ARM Cortex-A57), and 2GB RAM)
- Android OS v6.0.1 (Marshmallow) with Linux kernel v3.14 (64-bit)
- Changed bionic linker and linux kernel
- Tested for 20 core system binaries

System Modifications	Norax Components	SLoC	Language
Linux Kernel	NLoader, NMonitor	1947	C
Bionic Linker	NLoader	289	C++
Analysis & Rewriting Modules	NDisassembler, NPatcher	3580	Python & Bash Shell Script

Module	Size (Stock)	Size (NORAX)	File Size Overhead	# of Rewrite Errors
vold	486,032	512,736	5.49%	0
toybox	310,800	322,888	3.89%	0
toolbox	148,184	154,632	4.35%	0
dhcpcd	112,736	116,120	3.00%	0
logd	83,904	86,256	2.80%	0
installd	72,152	76,896	6.58%	0
app_process64 (zygote)	22,456	23,016	2.49%	0
qseecomd	14,584	15,032	3.07%	0
surfaceflinger	14,208	14,448	1.69%	0
rild	14,216	14,784	4.00%	0
libart.so	7,512,272	7,772,520	3.46%	0
libstagefright.so	1,883,288	1,946,328	3.35%	0
libcrypto.so	1,137,280	1,157,816	1.81%	0
libmedia.so	1,058,616	1,071,712	1.24%	0
libc.so	1,032,392	1,051,312	1.83%	0
libc++.so	944,056	951,632	0.80%	0
libsqlite.so	791,176	805,784	1.85%	0
libbinder.so	325,416	327,072	0.51%	0
libm.so	235,544	293,744	24.71%	0
libandroid.so	96,032	97,208	1.22%	0
AVG.			3.91%	0

Evaluation - functionality test

Module	Description	Experiment	Success
vold	Volume daemon	mount SDCard; umount	Yes
toybox	***115*** **nix utilities*	try all commands	Yes
toolbox	***22*** *core *nix utilities*	try all commands	Yes
dhcpcd	DHCP daemon	obtain dynamic IP address	Yes
logd	Logging daemon	collect system log for 1 hour	Yes
installd	APK install daemon	install 10 APKs	Yes
app_process64 (zygote)	Parent process for all applications	open 20 apps; close	Yes
qseecomd	Qualcomm's proprietary driver	boot up the phone	Yes
surfaceflinger	Compositing frame buffers for disply	Take 5 photos; play 30 min movie	Yes
rild	Baseband service daemon	Have 10 min phone call	Yes

Figure: Functionality Test Result

	Pass	Fail	Not Executed	Plan Name
CTS normal	126,457	552	0	CTS
CTS NORAX	126,457	552	0	CTS

Figure: Compatibility evaluation with Android Compatibility Test Suite (CTS)

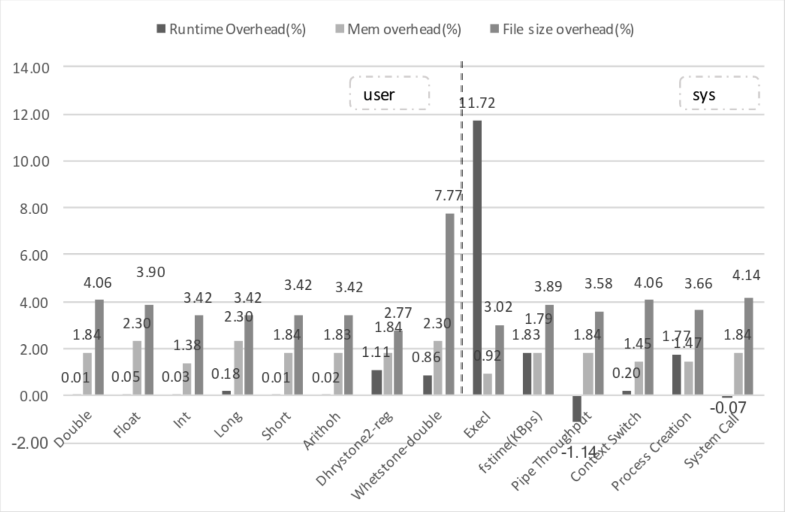
Evaluation - embedded data identification

- ground truth: compiled with debugging sections (dwarf .debug_*)
- very few gadgets in extracted inline data

Module	#. of Real Inline Data (Byte)	#. of Inline Data Flagged by Norax (Byte)	#. of Gadgets found in extracted inline Data
vold	0	0	0
toybox	8	8	0
toolbox	20	20	0
dhcpcd	40	40	4
Logd	0	0	0
installd	0	0	0
app_process64 (zygote)	0	0	0
qseecomd	N/A	0	0
surfaceflinger	0	0	0
rild	0	0	0
libart.so	17716	17716	8
libstagefright.so	296	296	5
libcrypto.so	2472	2512	25
libmedia.so	3936	3936	0
libc.so	4836	4836	5
libc++.so	12	12	0
libsqlite.so	932	1004	13
libbinder.so	0	0	0
libm.so	20283	20291	48
libandroid.so	0	0	0
Total	50551	50671	108

Evaluation - performance

- average performance overhead: **1.18%**
- average memory overhead: **2.21%**



Code Pointer?

- The address of next instruction after bl is stored on stack and visible to attacker
- Function pointer or function address in .got are visible to attacker

```c
#include <stdio.h>

void foo(void)
{
      printf("Hello World!\n");
}

int main(int argc, char **argv)
{
      foo();
      return 0;
}
```

```
00000000004005c0 <foo>:
  4005c0:   a9bf7bfd   stp   x29, x30, [sp,#-16]!
  4005c4:   910003fd   mov   x29, sp
  4005c8:   90000000   adrp  x0, 400000 <_init-0x3f0>
  4005cc:   911a6000   add   x0, x0, #0x698
  4005d0:   97ffffa4   bl    400460 <puts@plt>
  4005d4:   d503201f   nop
  4005d8:   a8c17bfd   ldp   x29, x30, [sp],#16
  4005dc:   d65f03c0   ret

00000000004005e0 <main>:
  4005e0:   a9be7bfd   stp   x29, x30, [sp,#-32]!
  4005e4:   910003fd   mov   x29, sp
  4005e8:   b9001fa0   str   w0, [x29,#28]
  4005ec:   f9000ba1   str   x1, [x29,#16]
  4005f0:   97fffff4   bl    4005c0 <foo>
  4005f4:   52800000   mov   w0, #0x0                   // #0
  4005f8:   a8c27bfd   ldp   x29, x30, [sp],#32
  4005fc:   d65f03c0   ret
```

change sp, then
store x29 (FP) and x30 (LR)

load x29 (FP) and x30 (LR),
then change sp

bl stores address of
next instruction to x30 (LR)

Interesting Paper/Link

- 64-bit Linux Return-Oriented Programming. http://crypto.stanford.edu/∼blynn/rop
- ROPgadget: https://github.com/JonathanSalwan/ROPgadget
- Practical Code Randomization Resilient to Memory Disclosure. IEEE S & P 2015
- Control Flow Integrity for COTS Binaries. USENIX Security 2013
- SoK: Eternal War in Memory. IEEE S & P 2013
- http://shell-storm.org
- Control-Flow Integrity. CCS 2005

Take-Home Message

- Shellcode injection and execution are not prerequisite for ROP

Take-Home Message

- Shellcode injection and execution are not prerequisite for ROP
- Fine-grained ASLR cannot defend JIT-ROP attack

Take-Home Message

- Shellcode injection and execution are not prerequisite for ROP
- Fine-grained ASLR cannot defend JIT-ROP attack
- Direct memory disclosure and indirect memory disclosure

Take-Home Message

- Shellcode injection and execution are not prerequisite for ROP
- Fine-grained ASLR cannot defend JIT-ROP attack
- Direct memory disclosure and indirect memory disclosure
- XOM is supported by Intel EPT and AArch64 userspace

Take-Home Message

- Shellcode injection and execution are not prerequisite for ROP
- Fine-grained ASLR cannot defend JIT-ROP attack
- Direct memory disclosure and indirect memory disclosure
- XOM is supported by Intel EPT and AArch64 userspace
- Code-data separation is possible for AArch64 COTS binary