Secure Software Development

Countermeasures: Exploitation Prevention & Privilege Minimization

Daniel Gruss, Vedad Hadzic, Lukas Maar, Stefan Gast, Marcel Nageler

02.12.2022

Winter 2023/24, www.iaik.tugraz.at

Table of contents

Defense In Depth

Attacker's perspective

- Vulnerability discovery ☑
- Exploitation ☑
- Privilege elevation ☑

Defender's perspective

- Vulnerability prevention ☑
- Exploit prevention (today)
- Privilege minimization (today)

Defense In Depth

Attacker's perspective

- Vulnerability discovery
  - buffer/integer overflow, use-after-free, format strings, type confusion
- Exploitation
  - Data corruption, shellcode, code reuse, ROP, return-to-libc
- Privilege elevation
  - exploit suid binaries, kernel exploits, crack root PW hash ;)

Defense In Depth

Attacker's perspective

- Vulnerability discovery
  - buffer/integer overflow, use-after-free, format strings, type confusion
- Exploitation
  - Data corruption, shellcode, code reuse, ROP, return-to-libc
- Privilege elevation
  - exploit suid binaries, kernel exploits, crack root PW hash ;)

Defender's perspective

- Vulnerability prevention
  - Code quality, memory safety, type safety, error handling ...
- Exploit prevention
  - Compiler/runtime defenses, hardware defenses
- Privilege minimization
  - System call filtering, sandboxing, virtualization

Recap: What if Vulnerability Prevention Fails?

- Attacker triggered a vulnerability

Recap: What if Vulnerability Prevention Fails?

- Attacker triggered a vulnerability
  - Part 1: Can we prevent exploitation?

Recap: What if Vulnerability Prevention Fails?

- Attacker triggered a vulnerability
  - Part 1: Can we prevent exploitation? → Exploit Prevention

Recap: What if Vulnerability Prevention Fails?

- Attacker triggered a vulnerability
  - Part 1: Can we prevent exploitation? → Exploit Prevention
- Attacker gained arbitrary code execution

Recap: What if Vulnerability Prevention Fails?

- Attacker triggered a vulnerability
  - Part 1: Can we prevent exploitation? → Exploit Prevention
- Attacker gained arbitrary code execution
  - Part 2: Can we prevent further damage?

Recap: What if Vulnerability Prevention Fails?

- Attacker triggered a vulnerability
  - Part 1: Can we prevent exploitation? → Exploit Prevention
- Attacker gained arbitrary code execution
  - Part 2: Can we prevent further damage? → Privilege Minimization

Recap: What if Vulnerability Prevention Fails?

- Attacker triggered a vulnerability
  - Part 1: Can we prevent exploitation? → Exploit Prevention
- Attacker gained arbitrary code execution
  - Part 2: Can we prevent further damage? → Privilege Minimization
- Defenses must be cheap!

Stack Buffer Overflows

Recap: Buffer overflow


```c
void printName(char* buffer) {
  char name[16];
  strcpy(name, buffer);
  printf("Hello %s\n", name);
}

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

Recap: Buffer overflow


```c
void printName(char* buffer) {
  char name[16];
  strcpy(name, buffer);
  printf("Hello %s\n", name);
}

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

👁 Observation 1: Buffer overflows are mostly linear

Recap: Buffer overflow


```
void printName(char* buffer) {
  char name[16];
  strcpy(name, buffer);
  printf("Hello %s\n", name);
}

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

- Observation 1: Buffer overflows are mostly linear
  - Cannot hit arbitrary memory, unlike format string vulnerabilities

Recap: Buffer overflow


```c
void printName(char* buffer) {
  char name[16];
  strcpy(name, buffer);
  printf("Hello %s\n", name);
}

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

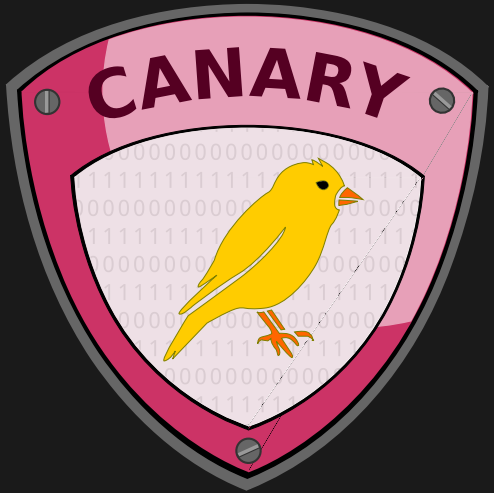
- Observation 1: Buffer overflows are mostly linear
  - Cannot hit arbitrary memory, unlike format string vulnerabilities
- Observation 2: Attackers typically overwrite code pointer (return address)

Recap: Buffer overflow


```c
void printName(char* buffer) {
  char name[16];
  strcpy(name, buffer);
  printf("Hello %s\n", name);
}

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

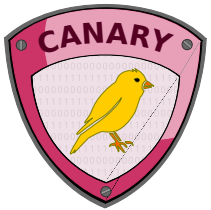
- Observation 1: Buffer overflows are mostly linear
  - Cannot hit arbitrary memory, unlike format string vulnerabilities
- Observation 2: Attackers typically overwrite code pointer (return address)
- How can we detect linear buffer overflows?


CANARY

Using Canaries for Safety

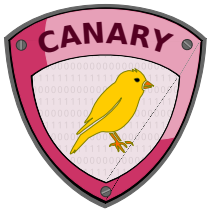
- If the mine canary is dead, get out immediately

Stack Canary I



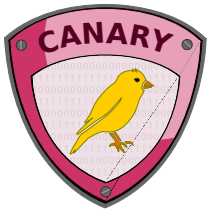
- Idea: Introduce a canary on the stack that signals a hazard

Stack Canary I



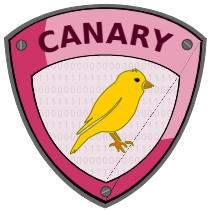
- Idea: Introduce a canary on the stack that signals a hazard
  - Hazard = corrupted return address

Stack Canary I



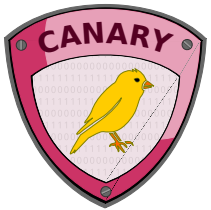
- Idea: Introduce a canary on the stack that signals a hazard
  - Hazard = corrupted return address
- Implementation

Stack Canary I



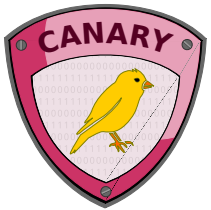
- Idea: Introduce a canary on the stack that signals a hazard
  - Hazard = corrupted return address
- Implementation
  - Simple compiler extension

Stack Canary I

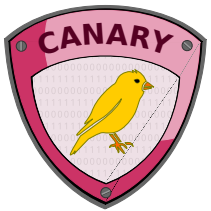


- Idea: Introduce a canary on the stack that signals a hazard
  - Hazard = corrupted return address
- Implementation
  - Simple compiler extension
  - Function prologue: push a random value (the canary), after the return address

Stack Canary I

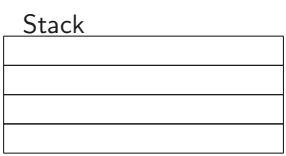


- Idea: Introduce a canary on the stack that signals a hazard
  - Hazard = corrupted return address
- Implementation
  - Simple compiler extension
  - Function prologue: push a random value (the canary), after the return address
  - Linear buffer overflow can only overwrite return address when also overwriting canary



- Idea: Introduce a canary on the stack that signals a hazard
  - Hazard = corrupted return address
- Implementation
  - Simple compiler extension
  - Function prologue: push a random value (the canary), after the return address
  - Linear buffer overflow can only overwrite return address when also overwriting canary
  - Function epilogue: check if canary is valid (unmodified) before doing `retq`

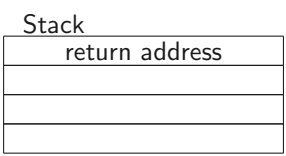
Stack Canary II




```
<func>:
  Setup stack frame
  PUSH canary
  [...]
  ; check canary
  RET

<main>:
  [...]
  CALL func
  [...]
```

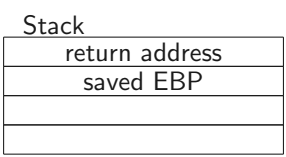
Stack Canary II




```
<func>:
   Setup stack frame
   PUSH canary
   [...]
   ; check canary
   RET

<main>:
   [...]
   CALL func
   [...]
```

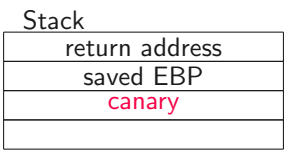
Stack Canary II




```
<func>:
  Setup stack frame
  PUSH canary
  [...]
  ; check canary
  RET

<main>:
  [...]
  CALL func
  [...]
```

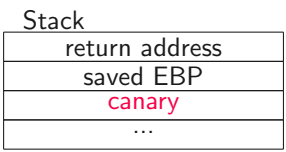
Stack Canary II




```
<func>:
  Setup stack frame
  PUSH canary
  [...]
  ; check canary
  RET

<main>:
  [...]
  CALL func
  [...]
```

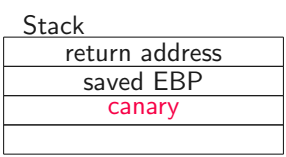
Stack Canary II




```
<func>:
  Setup stack frame
  PUSH canary
  [...]
  ; check canary
  RET

<main>:
  [...]
  CALL func
  [...]
```

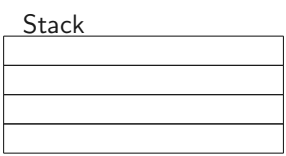
Stack Canary II




```
<func>:
  Setup stack frame
  PUSH canary
  [...]
  ; check canary
  RET

<main>:
  [...]
  CALL func
  [...]
```

Stack Canary II




```
<func>:
   Setup stack frame
   PUSH canary
   [...]
   ; check canary
   RET

<main>:
   [...]
   CALL func
   [...]
```

Stack Canary Example I

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

void printName(char* buffer) {
  char name[16];
  strcpy(name, buffer);
  printf("Hello %s\n", name);
}

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

Stack Canary Example II

```
% gcc -o stack -fno-stack-protector stack.c
```

Stack Canary Example II

```
% gcc -o stack -fno-stack-protector stack.c
% ./stack AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Stack Canary Example II

```
% gcc -o stack -fno-stack-protector stack.c
% ./stack AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[1]    12345 segmentation fault (core dumped)  ./stack
```

Stack Canary Example II

```
% gcc -o stack -fno-stack-protector stack.c
% ./stack AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[1]    12345 segmentation fault (core dumped)  ./stack
```

```
% gcc -o stack -fstack-protector stack.c
```

Stack Canary Example II

```
% gcc -o stack -fno-stack-protector stack.c
% ./stack AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[1]    12345 segmentation fault (core dumped)  ./stack
```

```
% gcc -o stack -fstack-protector stack.c
% ./stack AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Stack Canary Example II

```
% gcc -o stack -fno-stack-protector stack.c
% ./stack AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[1]    12345 segmentation fault (core dumped)  ./stack
```

```
% gcc -o stack -fstack-protector stack.c
% ./stack AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
*** stack smashing detected ***: ./stack terminated
```

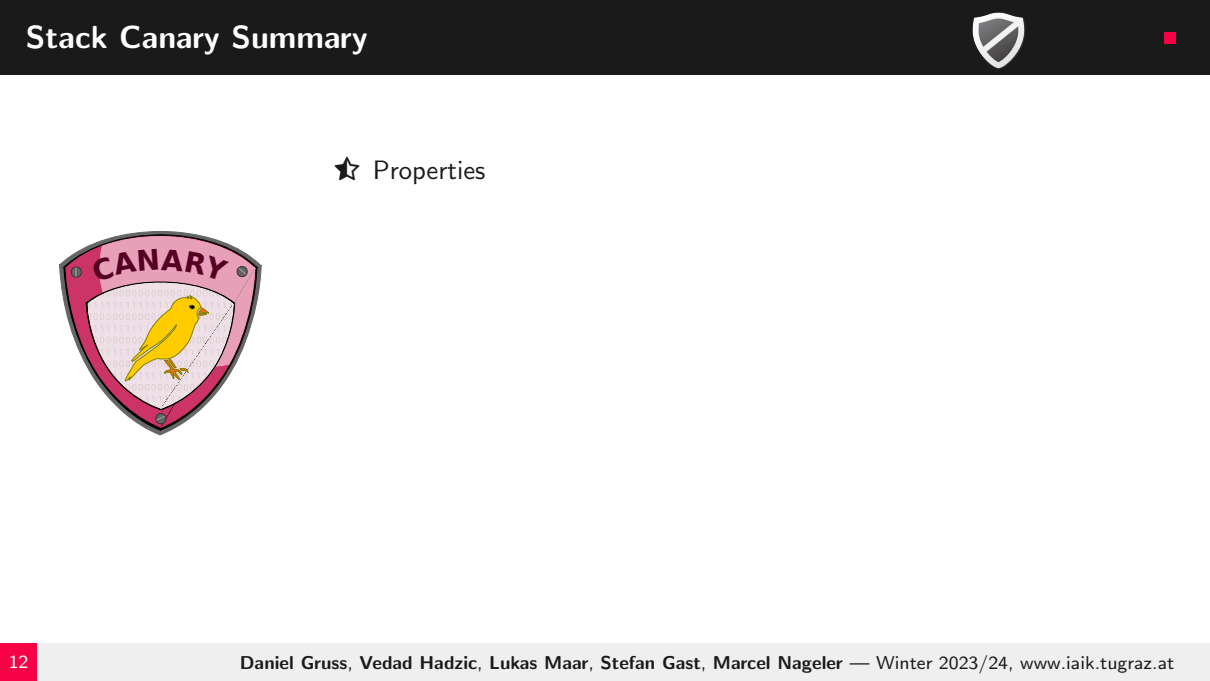
Stack Canary Example III

```
% objdump -d stack
```

Stack Canary Example III

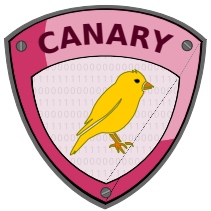
```
% objdump -d stack
00000000004005d6 <printName>:
  // function prologue
  ...
  4005e2: mov    %fs:0x28,%rax   // load canary value
  4005eb: mov    %rax,-0x8(%rbp) // store canary on stack
  4005ef: xor    %eax,%eax
  ...
  // function epilogue
  40061b: mov    -0x8(%rbp),%rax // load canary from stack
  40061f: xor    %fs:0x28,%rax   // compare
  400628: je     40062f <printName+0x59>
  40062a: callq  4004a0 <__stack_chk_fail@plt>
  40062f: leaveq
  400630: retq
```

Stack Canary Summary



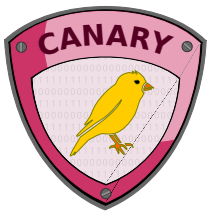
- Properties

Stack Canary Summary



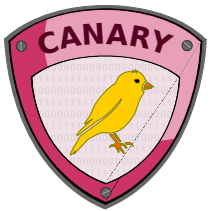
- Properties
  - Detects linear stack buffer overflows corrupting return address

Stack Canary Summary



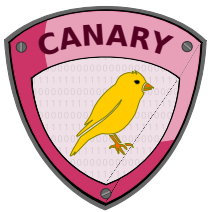
- Properties
  - Detects linear stack buffer overflows corrupting return address
  - Does not detect

Stack Canary Summary



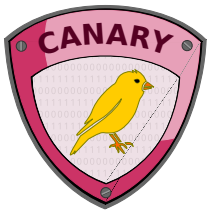
- Properties
  - Detects linear stack buffer overflows corrupting return address
  - Does not detect
    - overflowing one buffer into the other

Stack Canary Summary



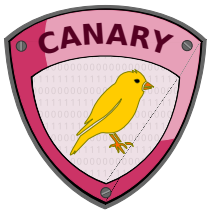
- Properties
  - Detects linear stack buffer overflows corrupting return address
  - Does not detect
    - overflowing one buffer into the other
    - overflowing co-located variables

Stack Canary Summary



- Properties
  - Detects linear stack buffer overflows corrupting return address
  - Does not detect
    - overflowing one buffer into the other
    - overflowing co-located variables
    - arbitrary write access, e.g. `buffer[input] = input2;`

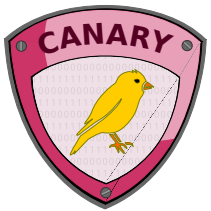
Stack Canary Summary



Properties

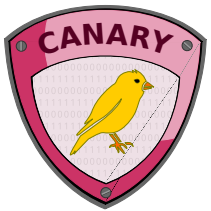
- Detects linear stack buffer overflows corrupting return address
- Does not detect
  - overflowing one buffer into the other
  - overflowing co-located variables
  - arbitrary write access, e.g. `buffer[input] = input2;`
  - format string vulnerabilities ...

Stack Canary Summary



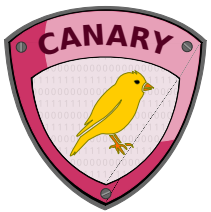
- Properties
  - Detects linear stack buffer overflows corrupting return address
  - Does not detect
    - overflowing one buffer into the other
    - overflowing co-located variables
    - arbitrary write access, e.g. `buffer[input] = input2;`
    - format string vulnerabilities ...
  - Probabilistic defense

Stack Canary Summary



- Properties
  - Detects linear stack buffer overflows corrupting return address
  - Does not detect
    - overflowing one buffer into the other
    - overflowing co-located variables
    - arbitrary write access, e.g. `buffer[input] = input2;`
    - format string vulnerabilities ...
  - Probabilistic defense
    - Ineffective if attacker can guess or leak the canary value

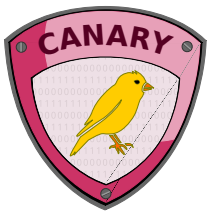
Stack Canary Summary



Properties

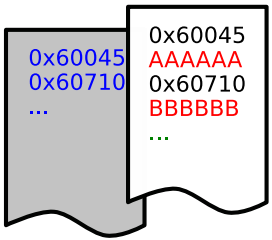
- Detects linear stack buffer overflows corrupting return address
- Does not detect
  - overflowing one buffer into the other
  - overflowing co-located variables
  - arbitrary write access, e.g. `buffer[input] = input2;`
  - format string vulnerabilities ...
- Probabilistic defense
  - Ineffective if attacker can guess or leak the canary value
- Animal welfare compatible (no bird has to die ;)

Stack Canary Summary

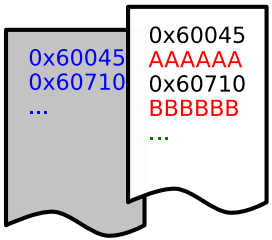


- Properties
  - Detects linear stack buffer overflows corrupting return address
  - Does not detect
    - overflowing one buffer into the other
    - overflowing co-located variables
    - arbitrary write access, e.g. `buffer[input] = input2;`
    - format string vulnerabilities ...
  - Probabilistic defense
    - Ineffective if attacker can guess or leak the canary value
  - Animal welfare compatible (no bird has to die ;)
- If we push/pop a canary for each return address, why not just duplicate return addresses instead?

Shadow Stack

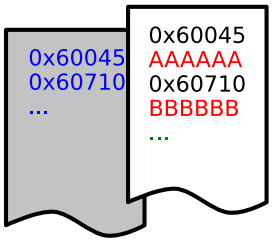


Shadow Stack



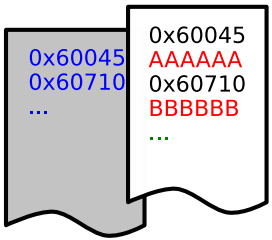
- Idea: duplicate return address on separate stack

Shadow Stack



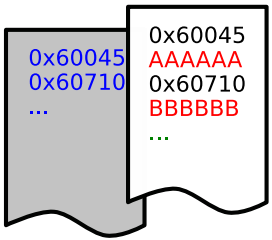
- Idea: duplicate return address on separate stack
- Implementation: compiler extension similar to canaries

Shadow Stack



- Idea: duplicate return address on separate stack
- Implementation: compiler extension similar to canaries
  - Prologue: push return address also on shadow stack

Shadow Stack



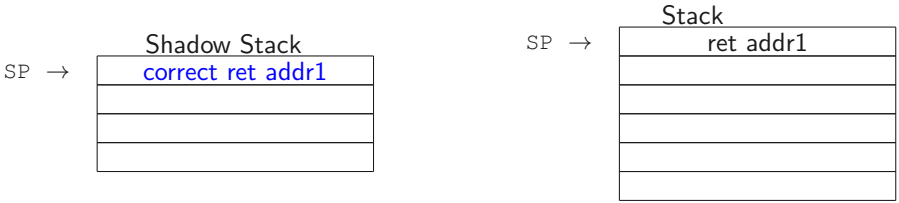
- Idea: duplicate return address on separate stack
- Implementation: compiler extension similar to canaries
  - Prologue: push return address also on shadow stack
  - Epilogue: verify return address before doing `retq`

Shadow Stack Example

Shadow Stack Example

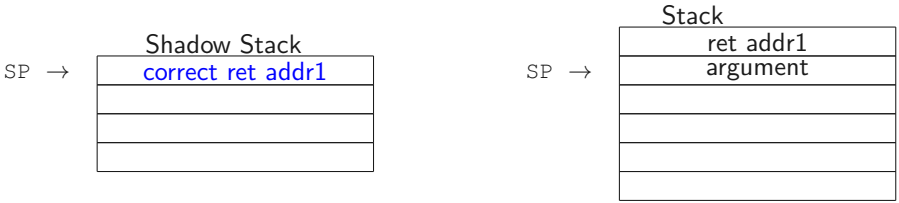
- Shadow stack duplicates all return addresses

Shadow Stack Example



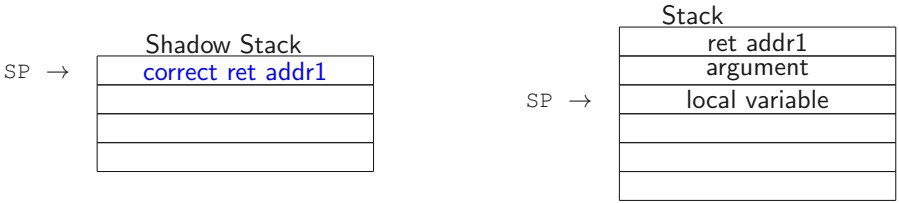
- Shadow stack duplicates all return addresses

Shadow Stack Example



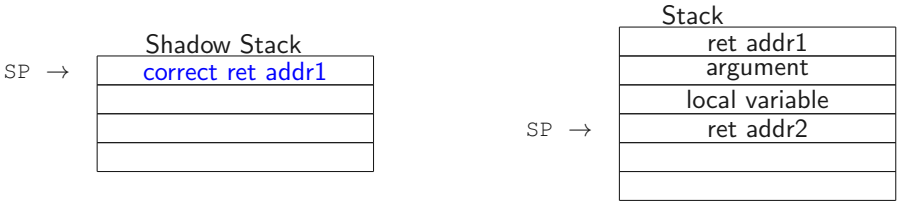
- Shadow stack duplicates all return addresses

Shadow Stack Example



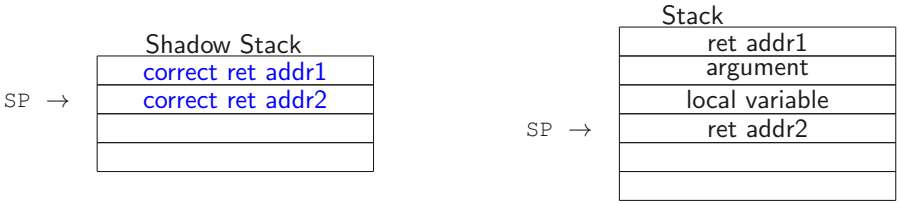
- Shadow stack duplicates all return addresses

Shadow Stack Example



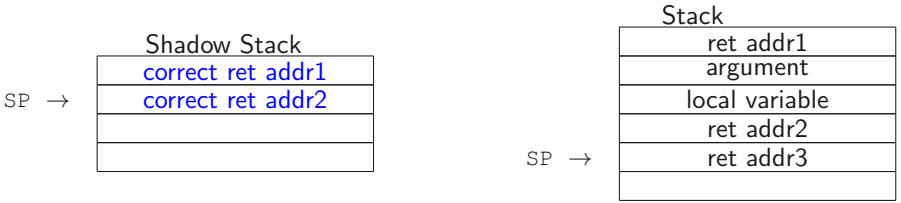
- Shadow stack duplicates all return addresses

Shadow Stack Example



- Shadow stack duplicates all return addresses

Shadow Stack Example



- Shadow stack duplicates all return addresses

Shadow Stack Example

- Shadow stack duplicates all return addresses

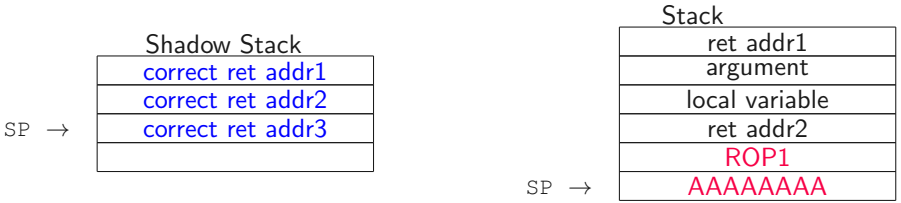
Shadow Stack Example

- Shadow stack duplicates all return addresses
- Attacker injects ROP chain

Shadow Stack Example

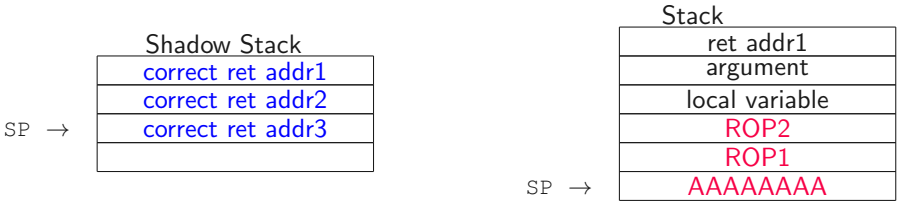
- Shadow stack duplicates all return addresses
- Attacker injects ROP chain

Shadow Stack Example



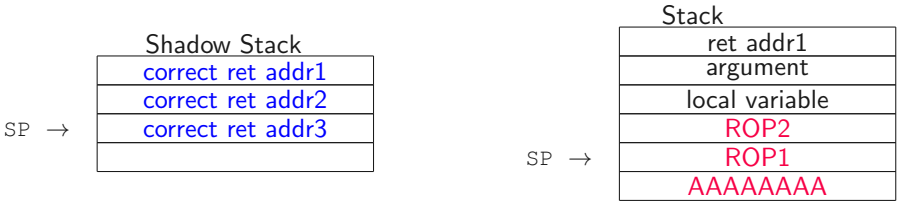
- Shadow stack duplicates all return addresses
- Attacker injects ROP chain

Shadow Stack Example



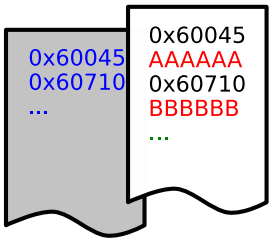
- Shadow stack duplicates all return addresses
- Attacker injects ROP chain

Shadow Stack Example



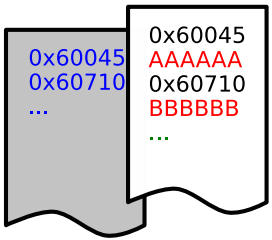
- Shadow stack duplicates all return addresses
- Attacker injects ROP chain
- Program crashes because of shadow stack mismatch

Shadow Stack Summary



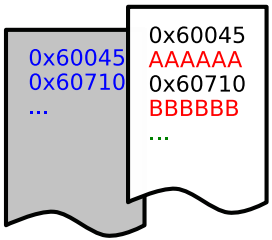
- Properties

Shadow Stack Summary



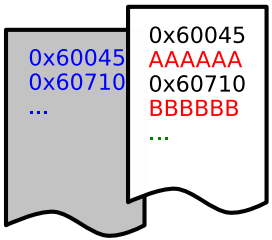
- Properties
  - Detect if buffer overflow corrupts return address

Shadow Stack Summary



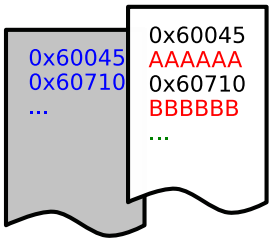
- Properties
  - Detect if buffer overflow corrupts return address
  - Does not prevent

Shadow Stack Summary



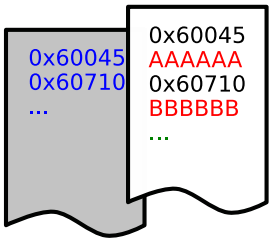
- Properties
  - Detect if buffer overflow corrupts return address
  - Does not prevent
    - attacks not affecting return addresses

Shadow Stack Summary



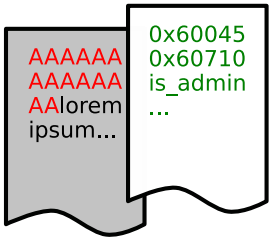
- Properties
  - Detect if buffer overflow corrupts return address
  - Does not prevent
    - attacks not affecting return addresses
    - attacks on shadow stack

Shadow Stack Summary

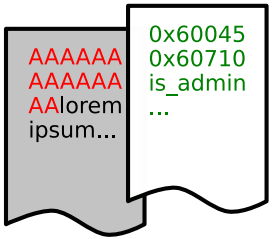


- Properties
  - Detect if buffer overflow corrupts return address
  - Does not prevent
    - attacks not affecting return addresses
    - attacks on shadow stack
- Why duplicate at all if we assume shadow stack is secure?

Safe Stack

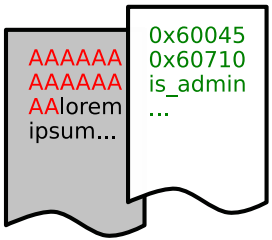


Safe Stack



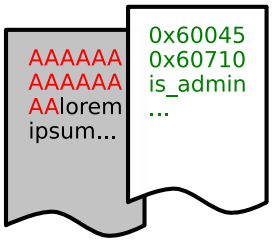
- Inverse idea: store unsafe buffers on separate stack

Safe Stack



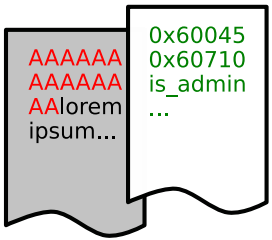
- Inverse idea: store unsafe buffers on separate stack
  - Safe stack only contains return addresses and sensible variables that cannot overflow

Safe Stack



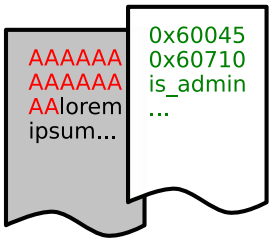
- Inverse idea: store unsafe buffers on separate stack
  - Safe stack only contains return addresses and sensible variables that cannot overflow
- Implementation: compiler extension similar to shadow stack

Safe Stack Summary



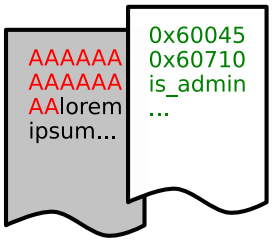
- Properties

Safe Stack Summary



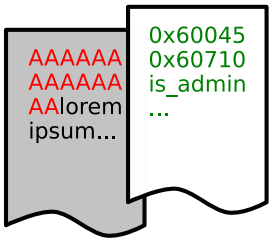
- Properties
  - Buffer overflow cannot corrupt return addresses / safe variables

Safe Stack Summary



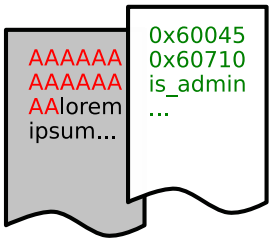
- Properties
  - Buffer overflow cannot corrupt return addresses / safe variables
  - Does not prevent

Safe Stack Summary



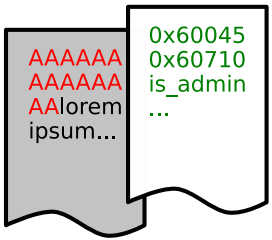
- Properties
  - Buffer overflow cannot corrupt return addresses / safe variables
  - Does not prevent
    - overflowing one unsafe buffer into the other

Safe Stack Summary



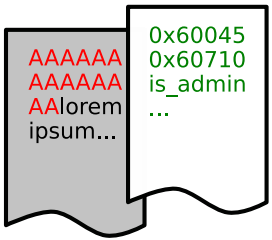
- Properties
  - Buffer overflow cannot corrupt return addresses / safe variables
  - Does not prevent
    - overflowing one unsafe buffer into the other
    - format string vulnerabilities ...

Safe Stack Summary



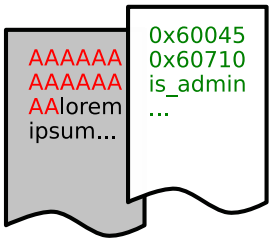
- Properties
  - Buffer overflow cannot corrupt return addresses / safe variables
  - Does not prevent
    - overflowing one unsafe buffer into the other
    - format string vulnerabilities ...
    - attacks on safe stack

Safe Stack Summary



- Properties
  - Buffer overflow cannot corrupt return addresses / safe variables
  - Does not prevent
    - overflowing one unsafe buffer into the other
    - format string vulnerabilities ...
    - attacks on safe stack
- Why not protect shadow/safe stack in hardware?

Safe Stack Summary



- Properties
  - Buffer overflow cannot corrupt return addresses / safe variables
  - Does not prevent
    - overflowing one unsafe buffer into the other
    - format string vulnerabilities ...
    - attacks on safe stack
- Why not protect shadow/safe stack in hardware?
  - Control-Flow Enforcement Technology (CET) for Intel (and AMD)

Code Injection Attacks

Recap: Shellcode

- Exploit buffer overflow

Recap: Shellcode

- Exploit buffer overflow
- Inject custom code that spawns a shell → Shellcode

Recap: Shellcode

- Exploit buffer overflow
- Inject custom code that spawns a shell → Shellcode
- Corrupt code pointer to execute shellcode

Data Execution Prevention

Data Execution Prevention (DEP)

Data Execution Prevention

Data Execution Prevention (DEP) $\approx$ Write-Xor-Execute (W$\oplus$X)

Data Execution Prevention

Data Execution Prevention (DEP) $\approx$ Write-Xor-Execute (W$\oplus$X)

👁 Observation: Von-Neumann CPUs mix code and data memory

- This allows code injection into data memory

Data Execution Prevention

Data Execution Prevention (DEP) ≈ Write-Xor-Execute (W⊕X)

- Observation: Von-Neumann CPUs mix code and data memory
  - This allows code injection into data memory
- Idea 1: make data memory non-executable

Data Execution Prevention

Data Execution Prevention (DEP) $\approx$ Write-Xor-Execute (W$\oplus$X)

- Observation: Von-Neumann CPUs mix code and data memory
  - This allows code injection into data memory
- Idea 1: make data memory non-executable
- Idea 2: make code memory non-writable

Data Execution Prevention

Data Execution Prevention (DEP) $\approx$ Write-Xor-Execute (W$\oplus$X)

- Observation: Von-Neumann CPUs mix code and data memory
  - This allows code injection into data memory
- Idea 1: make data memory non-executable
- Idea 2: make code memory non-writable
- Implementation

Data Execution Prevention

Data Execution Prevention (DEP) ≈ Write-Xor-Execute (W⊕X)

- Observation: Von-Neumann CPUs mix code and data memory
  - This allows code injection into data memory
- Idea 1: make data memory non-executable
- Idea 2: make code memory non-writable
- Implementation
  - Set writable memory to non-executable, e.g.: stack, heap, data, ...

Data Execution Prevention

Data Execution Prevention (DEP) ≈ Write-Xor-Execute (W⊕X)

- Observation: Von-Neumann CPUs mix code and data memory
  - This allows code injection into data memory
- Idea 1: make data memory non-executable
- Idea 2: make code memory non-writable
- Implementation
  - Set writable memory to non-executable, e.g.: stack, heap, data, ...
  - Usually done by the program loader (using `mmap`, `mprotect`)

Data Execution Prevention

Data Execution Prevention (DEP) ≈ Write-Xor-Execute (W⊕X)

- Observation: Von-Neumann CPUs mix code and data memory
  - This allows code injection into data memory
- Idea 1: make data memory non-executable
- Idea 2: make code memory non-writable
- Implementation
  - Set writable memory to non-executable, e.g.: stack, heap, data, ...
  - Usually done by the program loader (using `mmap`, `mprotect`)
  - Hardware support in the page tables

Data Execution Prevention

Data Execution Prevention (DEP) ≈ Write-Xor-Execute (W⊕X)

- Observation: Von-Neumann CPUs mix code and data memory
  - This allows code injection into data memory
- Idea 1: make data memory non-executable
- Idea 2: make code memory non-writable
- Implementation
  - Set writable memory to non-executable, e.g.: stack, heap, data, ...
  - Usually done by the program loader (using `mmap`, `mprotect`)
  - Hardware support in the page tables
    - Intel: XD-bit, AMD: NX-bit, ARM: XN-bit

Data Execution Prevention Example I

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

char code[] = "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\
    x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05";

int main()
{
    printf("len:%d bytes\n", strlen(code));
    (*(void(*)()) code)();
    return 0;
}
```

Data Execution Prevention Example II

```
% gdb ./shellcode
```

Data Execution Prevention Example II

```
% gdb ./shellcode
(gdb) run
```

Data Execution Prevention Example II

```
% gdb ./shellcode
(gdb) run
Starting program: /home/shellcode
len:27 bytes

Program received signal SIGSEGV, Segmentation fault.
0x0000000000601040 in code ()
```

Data Execution Prevention Example II

```
% gdb ./shellcode
(gdb) run
Starting program: /home/shellcode
len:27 bytes

Program received signal SIGSEGV, Segmentation fault.
0x0000000000601040 in code ()
```

```
% execstack -s ./shellcode
```

Data Execution Prevention Example II

```
% gdb ./shellcode
(gdb) run
Starting program: /home/shellcode
len:27 bytes

Program received signal SIGSEGV, Segmentation fault.
0x0000000000601040 in code ()
```

```
% execstack -s ./shellcode
% gdb ./shellcode
```

Data Execution Prevention Example II

```
% gdb ./shellcode
(gdb) run
Starting program: /home/shellcode
len:27 bytes

Program received signal SIGSEGV, Segmentation fault.
0x0000000000601040 in code ()
```

```
% execstack -s ./shellcode
% gdb ./shellcode
(gdb) run
```

Data Execution Prevention Example II

```
% gdb ./shellcode
(gdb) run
Starting program: /home/shellcode
len:27 bytes

Program received signal SIGSEGV, Segmentation fault.
0x0000000000601040 in code ()
```

```
% execstack -s ./shellcode
% gdb ./shellcode
(gdb) run
Starting program: /home/shellcode
len:27 bytes
process 9494 is executing new program: /bin/dash
$
```

Data Execution Prevention Summary

- Properties of DEP/W⊕X

Data Execution Prevention Summary

- Properties of DEP/W⊕X
  - Prevents code injection attacks

Data Execution Prevention Summary

- Properties of DEP/W⊕X
  - Prevents code injection attacks
  - Hardly any runtime overhead

Data Execution Prevention Summary

- Properties of DEP/W⊕X
  - Prevents code injection attacks
  - Hardly any runtime overhead
  - Requires hardware support

Data Execution Prevention Summary

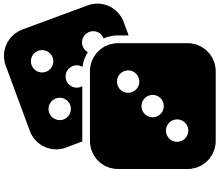
- Properties of DEP/W⊕X
  - Prevents code injection attacks
  - Hardly any runtime overhead
  - Requires hardware support
  - Does not prevent code reuse attacks (since no code is injected)

Data Execution Prevention Summary

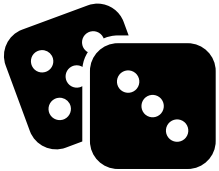
- Properties of DEP/W⊕X
  - Prevents code injection attacks
  - Hardly any runtime overhead
  - Requires hardware support
  - Does not prevent code reuse attacks (since no code is injected)
- How to protect just-in-time (JIT) compiled code? JIT compiler needs to modify code at runtime ...

Code Reuse Attacks

Randomization-based Defenses

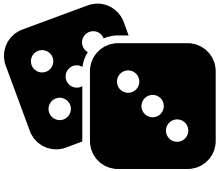


Randomization-based Defenses



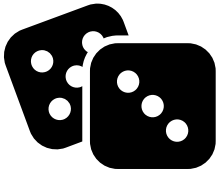
- Observation: Many exploits need knowledge of addresses (ROP, ret2libc ...)

Randomization-based Defenses



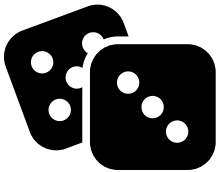
- Observation: Many exploits need knowledge of addresses (ROP, ret2libc ...)
- Idea: randomize program to make exploit development (much) more difficult

Randomization-based Defenses



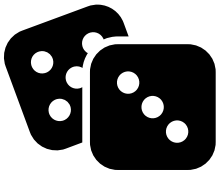
- Observation: Many exploits need knowledge of addresses (ROP, ret2libc ...)
- Idea: randomize program to make exploit development (much) more difficult
- General properties

Randomization-based Defenses



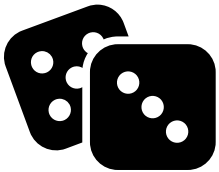
- Observation: Many exploits need knowledge of addresses (ROP, ret2libc ...)
- Idea: randomize program to make exploit development (much) more difficult
- General properties
  - Probabilistic defense

Randomization-based Defenses



- Observation: Many exploits need knowledge of addresses (ROP, ret2libc ...)
- Idea: randomize program to make exploit development (much) more difficult
- General properties
  - Probabilistic defense
    - Can be broken by information leakage (e.g., via side channels)

Randomization-based Defenses



- Observation: Many exploits need knowledge of addresses (ROP, ret2libc ...)
- Idea: randomize program to make exploit development (much) more difficult
- General properties
  - Probabilistic defense
    - Can be broken by information leakage (e.g., via side channels)
    - How big is the entropy?


ASLR

Address Space Layout Randomization (ASLR)

- Randomize the memory layout

Address Space Layout Randomization (ASLR)

- Randomize the memory layout
  - Attacker cannot guess location of libc, stack, heap ...

Address Space Layout Randomization (ASLR)

- Randomize the memory layout
  - Attacker cannot guess location of libc, stack, heap ...
- Implementation

Address Space Layout Randomization (ASLR)

- Randomize the memory layout
  - Attacker cannot guess location of libc, stack, heap ...
- Implementation
  - At program startup move various segments to a random position

Address Space Layout Randomization (ASLR)

- Randomize the memory layout
  - Attacker cannot guess location of libc, stack, heap ...
- Implementation
  - At program startup move various segments to a random position
    - Stack, heap, shared memory

Address Space Layout Randomization (ASLR)

- Randomize the memory layout
  - Attacker cannot guess location of libc, stack, heap ...
- Implementation
  - At program startup move various segments to a random position
    - Stack, heap, shared memory
    - Shared libraries

Address Space Layout Randomization (ASLR)

- Randomize the memory layout
  - Attacker cannot guess location of libc, stack, heap ...
- Implementation
  - At program startup move various segments to a random position
    - Stack, heap, shared memory
    - Shared libraries
    - Main executable (optional)

Address Space Layout Randomization (ASLR)

- Randomize the memory layout
  - Attacker cannot guess location of libc, stack, heap ...
- Implementation
  - At program startup move various segments to a random position
    - Stack, heap, shared memory
    - Shared libraries
    - Main executable (optional)
  - Randomization done by operating system (e.g., on `mmap`)

Address Space Layout Randomization (ASLR)

- Randomize the memory layout
  - Attacker cannot guess location of libc, stack, heap ...
- Implementation
  - At program startup move various segments to a random position
    - Stack, heap, shared memory
    - Shared libraries
    - Main executable (optional)
  - Randomization done by operating system (e.g., on `mmap`)
    - Linux `/proc/sys/kernel/randomize_va_space`

ASLR on Stack and Heap

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

int main() {
  int x;
  printf("Stack: %p\n", &x);
  printf("Heap:  %p\n", malloc(10));
  return 0;
}
```

ASLR on Stack and Heap

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

int main() {
  int x;
  printf("Stack: %p\n", &x);
  printf("Heap:  %p\n", malloc(10));
  return 0;
}
```

```
% ./aslr
Stack: 0x7ffcc2666e74
Heap:  0x1dd9420
```

ASLR on Stack and Heap

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

int main() {
  int x;
  printf("Stack: %p\n", &x);
  printf("Heap:  %p\n", malloc(10));
  return 0;
}
```

```
% ./aslr
Stack: 0x7ffcc2666e74
Heap:  0x1dd9420
% ./aslr
Stack: 0x7ffcbf0c1ae4
Heap:  0x124b420
```

ASLR on Stack, Heap, and Libraries

```
% cat /proc/self/maps
```

ASLR on Stack, Heap, and Libraries

```
% cat /proc/self/maps
00400000-0040c000 r-xp 00000000 fd:00 395191            /bin/cat
0060b000-0060c000 r--p 0000b000 fd:00 395191            /bin/cat
0060c000-0060d000 rw-p 0000c000 fd:00 395191            /bin/cat
00b0c000-00b2d000 rw-p 00000000 00:00 0                 [heap]
7efcbb558000-7efcbb87e000 r--p 00000000 fd:00 11534857  /usr/lib/locale/locale-archive
7efcbb87e000-7efcbba3e000 r-xp 00000000 fd:00 4587769   /lib/x86_64-linux-gnu/libc-2.23.so
7efcbba3e000-7efcbbc3e000 ---p 001c0000 fd:00 4587769   /lib/x86_64-linux-gnu/libc-2.23.so
7efcbbc3e000-7efcbbc42000 r--p 001c0000 fd:00 4587769   /lib/x86_64-linux-gnu/libc-2.23.so
7efcbbc42000-7efcbbc44000 rw-p 001c4000 fd:00 4587769   /lib/x86_64-linux-gnu/libc-2.23.so
7efcbbc44000-7efcbbc48000 rw-p 00000000 00:00 0
7efcbbc48000-7efcbbc6e000 r-xp 00000000 fd:00 4588089   /lib/x86_64-linux-gnu/ld-2.23.so
7efcbbe38000-7efcbbe3b000 rw-p 00000000 00:00 0
7efcbbe4b000-7efcbbe6d000 rw-p 00000000 00:00 0
7efcbbe6d000-7efcbbe6e000 r--p 00025000 fd:00 4588089   /lib/x86_64-linux-gnu/ld-2.23.so
7efcbbe6e000-7efcbbe6f000 rw-p 00026000 fd:00 4588089   /lib/x86_64-linux-gnu/ld-2.23.so
7efcbbe6f000-7efcbbe70000 rw-p 00000000 00:00 0
7ffff84c6000-7ffff84e7000 rw-p 00000000 00:00 0         [stack]
7ffff8536000-7ffff8538000 r--p 00000000 00:00 0         [vvar]
7ffff8538000-7ffff853a000 r-xp 00000000 00:00 0         [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

ASLR on Stack, Heap, Libraries

```
% cat /proc/self/maps
00400000-0040c000 r-xp 00000000 fd:00 395191                             /bin/cat
0060b000-0060c000 r--p 0000b000 fd:00 395191                             /bin/cat
0060c000-0060d000 rw-p 0000c000 fd:00 395191                             /bin/cat
00799000-007ba000 rw-p 00000000 00:00 0                                  [heap]
7fec1f08d000-7fec1f3b3000 r--p 00000000 fd:00 11534857                   /usr/lib/locale/locale-archive
7fec1f3b3000-7fec1f573000 r-xp 00000000 fd:00 4587769                    /lib/x86_64-linux-gnu/libc-2.23.so
7fec1f573000-7fec1f773000 ---p 001c0000 fd:00 4587769                    /lib/x86_64-linux-gnu/libc-2.23.so
7fec1f773000-7fec1f777000 r--p 001c0000 fd:00 4587769                    /lib/x86_64-linux-gnu/libc-2.23.so
7fec1f777000-7fec1f779000 rw-p 001c4000 fd:00 4587769                    /lib/x86_64-linux-gnu/libc-2.23.so
7fec1f779000-7fec1f77d000 rw-p 00000000 00:00 0
7fec1f77d000-7fec1f7a3000 r-xp 00000000 fd:00 4588089                    /lib/x86_64-linux-gnu/ld-2.23.so
7fec1f96d000-7fec1f970000 rw-p 00000000 00:00 0
7fec1f980000-7fec1f9a2000 rw-p 00000000 00:00 0
7fec1f9a2000-7fec1f9a3000 r--p 00025000 fd:00 4588089                    /lib/x86_64-linux-gnu/ld-2.23.so
7fec1f9a3000-7fec1f9a4000 rw-p 00026000 fd:00 4588089                    /lib/x86_64-linux-gnu/ld-2.23.so
7fec1f9a4000-7fec1f9a5000 rw-p 00000000 00:00 0
7ffeffa30000-7ffeffa51000 rw-p 00000000 00:00 0                          [stack]
7ffeffa7f000-7ffeffa81000 r--p 00000000 00:00 0                          [vvar]
7ffeffa81000-7ffeffa83000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

ASLR on Shared Libraries

- Same library code is randomized to different addresses at each program start

ASLR on Shared Libraries

- Same library code is randomized to different addresses at each program start

❓ How does randomized code remain functional?

ASLR on Code Requires Position Independence

- Within a module

ASLR on Code Requires Position Independence

- Within a module
  - Compiler replaces absolute addresses with (rip-)relative addresses
  - Code can be executed from virtually any offset
  - Shared libraries: compile flags `-fpic`, `-fPIC`
  - Executable: compile flags `-fpie`, `-fPIE`

ASLR on Code Requires Position Independence

- Within a module
  - Compiler replaces absolute addresses with (rip-)relative addresses
  - Code can be executed from virtually any offset
  - Shared libraries: compile flags `-fpic`, `-fPIC`
  - Executable: compile flags `-fpie`, `-fPIE`
- Across modules

ASLR on Code Requires Position Independence

- Within a module
  - Compiler replaces absolute addresses with (rip-)relative addresses
  - Code can be executed from virtually any offset
  - Shared libraries: compile flags `-fpic`, `-fPIC`
  - Executable: compile flags `-fpie`, `-fPIE`
- Across modules
  - Runtime linker resolves addresses via:
  - Global Offset Table (GOT) for arbitrary addresses
  - Procedure Linkage Table (PLT) for function calls

Executable Without Position Independence

```c
#include <stdio.h>

int main() {
  printf("Hi\n");
}
```

Executable Without Position Independence

```c
#include <stdio.h>

int main() {
  printf("Hi\n");
}
```

```
% gcc -o main -static main.c
```

Executable Without Position Independence

```c
#include <stdio.h>

int main() {
  printf("Hi\n");
}
```

```
% gcc -o main -static main.c
% objdump -d main
```

Executable Without Position Independence

```c
#include <stdio.h>

int main() {
  printf("Hi\n");
}
```

```
% gcc -o main -static main.c
% objdump -d main
00000000004009ae <main>:
  4009ae: 55                    push   %rbp
  4009af: 48 89 e5              mov    %rsp,%rbp
  4009b2: bf a4 11 4a 00        mov    $0x4a11a4,%edi    # Address of Hi
  4009b7: e8 24 f2 00 00        callq  40fbe0 <_IO_puts> # Direct call
  4009bc: b8 00 00 00 00        mov    $0x0,%eax
  4009c1: 5d                    pop    %rbp
  4009c2: c3                    retq
```

Executable With Position Independence

```c
#include <stdio.h>

int main() {
  printf("Hi\n");
}
```

```
% gcc -o main -fPIE main.c
```

Executable With Position Independence

```c
#include <stdio.h>

int main() {
  printf("Hi\n");
}
```

```
% gcc -o main -fPIE main.c
% objdump -d main
```

Executable With Position Independence

```c
#include <stdio.h>

int main() {
  printf("Hi\n");
}
```

```
% gcc -o main -fPIE main.c
% objdump -d main
0000000000000526 <main>:
     526: 55                    push   %rbp
     527: 48 89 e5              mov    %rsp,%rbp
     52a: 48 8d 3d 93 00 00 00  lea    0x93(%rip),%rdi   # Address of Hi
     531: e8 ca fe ff ff        callq  400 <puts@plt> # PLT
     536: b8 00 00 00 00        mov    $0x0,%eax
     53b: 5d                    pop    %rbp
     53c: c3                    retq
```

Address Space Layout Randomization (ASLR) Summary

- Properties

Address Space Layout Randomization (ASLR) Summary

- Properties
  - Cheap

Address Space Layout Randomization (ASLR) Summary

- Properties
  - Cheap
  - Make exploit development harder

Address Space Layout Randomization (ASLR) Summary

- Properties
  - Cheap
  - Make exploit development harder
  - Does not prevent exploitation within a module

Address Space Layout Randomization (ASLR) Summary

- Properties
  - Cheap
  - Make exploit development harder
  - Does not prevent exploitation within a module
  - Requires operating system support

Address Space Layout Randomization (ASLR) Summary

- Properties
  - Cheap
  - Make exploit development harder
  - Does not prevent exploitation within a module
  - Requires operating system support
  - ASLR on code requires position independence

Address Space Layout Randomization (ASLR) Summary

- Properties
  - Cheap
  - Make exploit development harder
  - Does not prevent exploitation within a module
  - Requires operating system support
  - ASLR on code requires position independence
  - Quite limited entropy on 32-bit systems

Address Space Layout Randomization (ASLR) Summary

- Properties
  - Cheap
  - Make exploit development harder
  - Does not prevent exploitation within a module
  - Requires operating system support
  - ASLR on code requires position independence
  - Quite limited entropy on 32-bit systems → brute-force

Address Space Layout Randomization (ASLR) Summary

- Properties
  - Cheap
  - Make exploit development harder
  - Does not prevent exploitation within a module
  - Requires operating system support
  - ASLR on code requires position independence
  - Quite limited entropy on 32-bit systems → brute-force
  - Information leak breaks ASLR

Address Space Layout Randomization (ASLR) Summary

- Properties
  - Cheap
  - Make exploit development harder
  - Does not prevent exploitation within a module
  - Requires operating system support
  - ASLR on code requires position independence
  - Quite limited entropy on 32-bit systems → brute-force
  - Information leak breaks ASLR
- How to protect ASLR against information leakage?

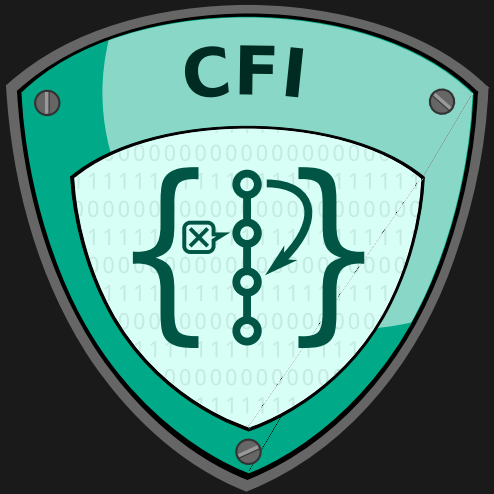
Address Space Layout Randomization (ASLR) Summary

- Properties
  - Cheap
  - Make exploit development harder
  - Does not prevent exploitation within a module
  - Requires operating system support
  - ASLR on code requires position independence
  - Quite limited entropy on 32-bit systems → brute-force
  - Information leak breaks ASLR
- How to protect ASLR against information leakage?
  - Execute-only memory (non-readable)

Challenge #6 - Selective ASLR

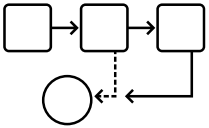
Change the randomization of the code segment

- You should generate two binaries with ASLR enabled
- One binary should have randomization for stack, heap, and code
- The other binary should only have randomization for stack and heap, but not for code
- Both binaries must run for at least 5 seconds (e.g., `sleep(5);` before return) but not longer than 10 seconds
- Upload your binaries at `https://challenges.sasectf.student.iaik.tugraz.at/aslr/index.php`
- If it is correct, you will get the flag
- Test system is Debian 11.8, kernel 5.10.0-26


CFI

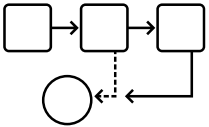
Control Flow Integrity (CFI)

- Observation: most attacks corrupt code pointers

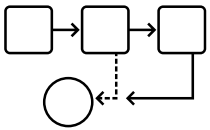


Control Flow Integrity (CFI)

- Observation: most attacks corrupt code pointers
- Idea: specifically protect code pointers

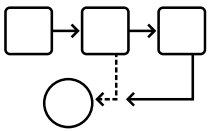


Control Flow Integrity (CFI)



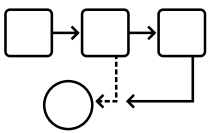
- Observation: most attacks corrupt code pointers
- Idea: specifically protect code pointers
  - CFI: Program must stay inside its approximated Control Flow Graph (CFG)

Control Flow Integrity (CFI)



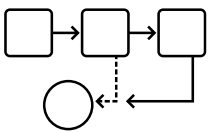
- Observation: most attacks corrupt code pointers
- Idea: specifically protect code pointers
  - CFI: Program must stay inside its approximated Control Flow Graph (CFG)
  - Attacker cannot (arbitrarily) change control flow

Control Flow Integrity (CFI)



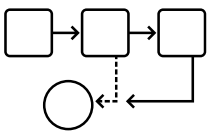
- Observation: most attacks corrupt code pointers
- Idea: specifically protect code pointers
  - CFI: Program must stay inside its approximated Control Flow Graph (CFG)
  - Attacker cannot (arbitrarily) change control flow
  - Prevent ret2libc, ROP, JOP, ...

Control Flow Integrity (CFI)



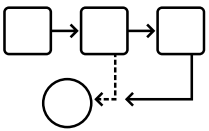
- Observation: most attacks corrupt code pointers
- Idea: specifically protect code pointers
  - CFI: Program must stay inside its approximated Control Flow Graph (CFG)
  - Attacker cannot (arbitrarily) change control flow
  - Prevent ret2libc, ROP, JOP, ...
- Code pointers everywhere

Control Flow Integrity (CFI)



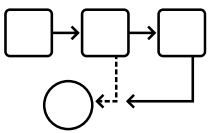
- Observation: most attacks corrupt code pointers
- Idea: specifically protect code pointers
  - CFI: Program must stay inside its approximated Control Flow Graph (CFG)
  - Attacker cannot (arbitrarily) change control flow
  - Prevent ret2libc, ROP, JOP, ...
- Code pointers everywhere
  - Return addresses

Control Flow Integrity (CFI)



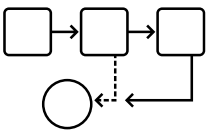
- Observation: most attacks corrupt code pointers
- Idea: specifically protect code pointers
  - CFI: Program must stay inside its approximated Control Flow Graph (CFG)
  - Attacker cannot (arbitrarily) change control flow
  - Prevent ret2libc, ROP, JOP, ...
- Code pointers everywhere
  - Return addresses
  - Function pointers

Control Flow Integrity (CFI)



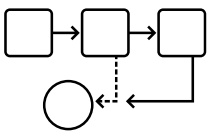
- Observation: most attacks corrupt code pointers
- Idea: specifically protect code pointers
  - CFI: Program must stay inside its approximated Control Flow Graph (CFG)
  - Attacker cannot (arbitrarily) change control flow
  - Prevent ret2libc, ROP, JOP, ...
- Code pointers everywhere
  - Return addresses
  - Function pointers
  - C++ vtables

Control Flow Integrity (CFI)



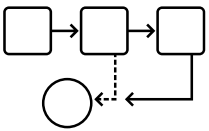
- Observation: most attacks corrupt code pointers
- Idea: specifically protect code pointers
  - CFI: Program must stay inside its approximated Control Flow Graph (CFG)
  - Attacker cannot (arbitrarily) change control flow
  - Prevent ret2libc, ROP, JOP, ...
- Code pointers everywhere
  - Return addresses
  - Function pointers
  - C++ vtables
  - GOT entries

Control Flow Integrity (CFI)



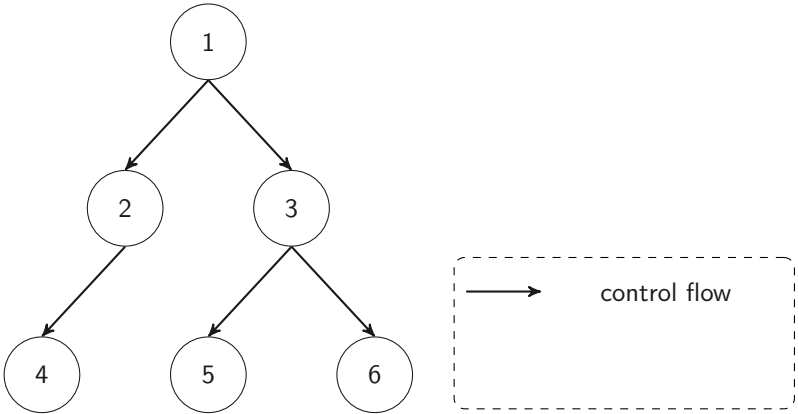
- Observation: most attacks corrupt code pointers
- Idea: specifically protect code pointers
  - CFI: Program must stay inside its approximated Control Flow Graph (CFG)
  - Attacker cannot (arbitrarily) change control flow
  - Prevent ret2libc, ROP, JOP, ...
- Code pointers everywhere
  - Return addresses
  - Function pointers
  - C++ vtables
  - GOT entries
  - Signal handlers

Control Flow Integrity (CFI)

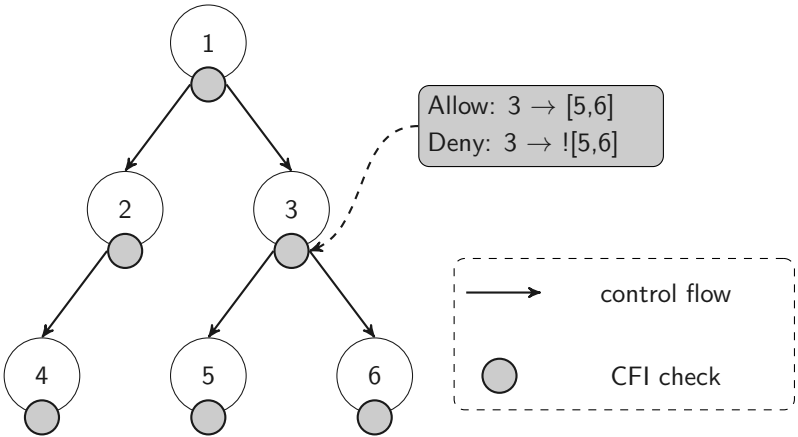


- Observation: most attacks corrupt code pointers
- Idea: specifically protect code pointers
  - CFI: Program must stay inside its approximated Control Flow Graph (CFG)
  - Attacker cannot (arbitrarily) change control flow
  - Prevent ret2libc, ROP, JOP, ...
- Code pointers everywhere
  - Return addresses
  - Function pointers
  - C++ vtables
  - GOT entries
  - Signal handlers
- We need to protect all of them!

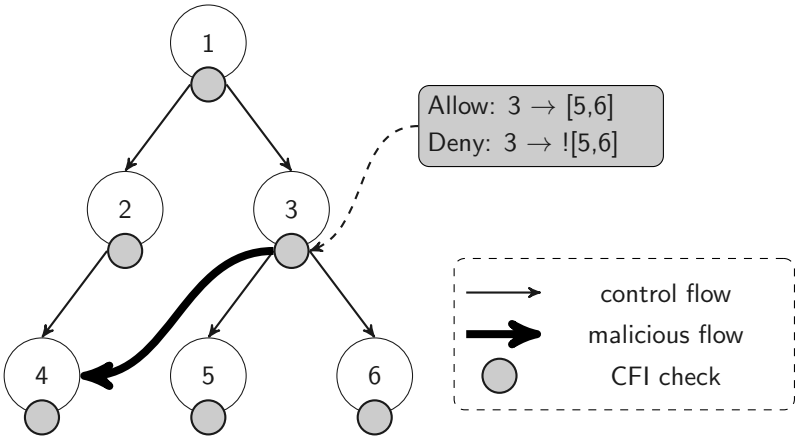
Control Flow Integrity (CFI)



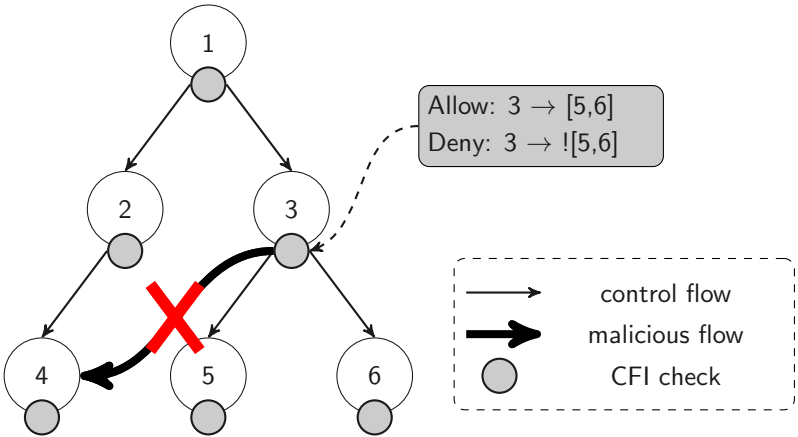
Control Flow Integrity (CFI)

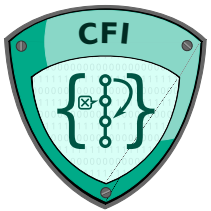


Control Flow Integrity (CFI)

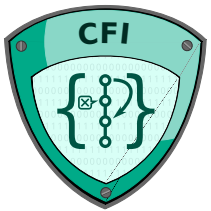


Control Flow Integrity (CFI)

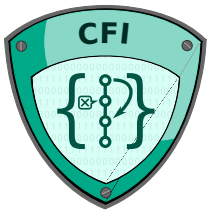




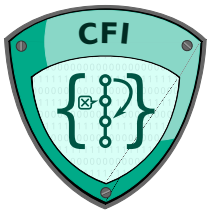
- CFI on backward edges



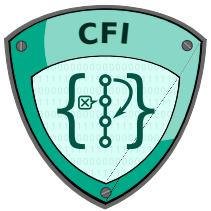
- CFI on backward edges
  - Return addresses shall only point to the real caller



- CFI on backward edges
  - Return addresses shall only point to the real caller
  - Solved via shadow stack / safe stack

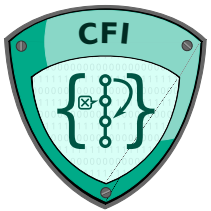


- CFI on backward edges
  - Return addresses shall only point to the real caller
  - Solved via shadow stack / safe stack
- CFI on forward edges

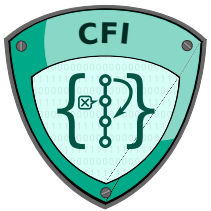


- CFI on backward edges
  - Return addresses shall only point to the real caller
  - Solved via shadow stack / safe stack
- CFI on forward edges
  - GOT entries shall only contain legitimate pointers

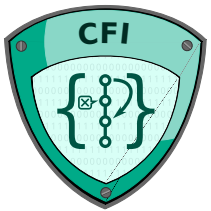
Control Flow Integrity (CFI)



- CFI on backward edges
  - Return addresses shall only point to the real caller
  - Solved via shadow stack / safe stack
- CFI on forward edges
  - GOT entries shall only contain legitimate pointers
  - Function pointers shall only call legitimate functions

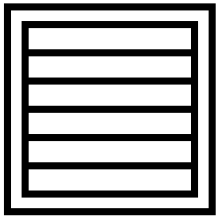


- CFI on backward edges
  - Return addresses shall only point to the real caller
  - Solved via shadow stack / safe stack
- CFI on forward edges
  - GOT entries shall only contain legitimate pointers
  - Function pointers shall only call legitimate functions
  - C++ vtable shall only call legitimate members



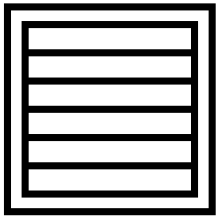
- CFI on backward edges
  - Return addresses shall only point to the real caller
  - Solved via shadow stack / safe stack
- CFI on forward edges
  - GOT entries shall only contain legitimate pointers
  - Function pointers shall only call legitimate functions
  - C++ vtable shall only call legitimate members
  - ❓ How to determine *legitimate* targets?

Global Offset Table (GOT) Protection



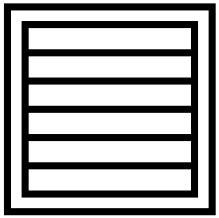
- Observation: only runtime linker populates GOT

Global Offset Table (GOT) Protection



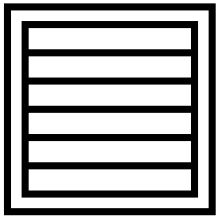
- Observation: only runtime linker populates GOT
  - Any other GOT manipulation is either an accident or an attack

Global Offset Table (GOT) Protection



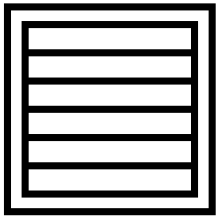
- Observation: only runtime linker populates GOT
  - Any other GOT manipulation is either an accident or an attack
- Idea: Make GOT read-only

Global Offset Table (GOT) Protection



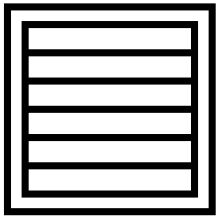
- Observation: only runtime linker populates GOT
  - Any other GOT manipulation is either an accident or an attack
- Idea: Make GOT read-only
- Implementation

Global Offset Table (GOT) Protection



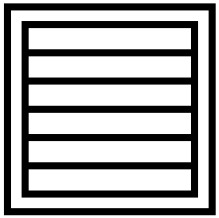
- Observation: only runtime linker populates GOT
  - Any other GOT manipulation is either an accident or an attack
- Idea: Make GOT read-only
- Implementation
  - Linker populates *all* GOT entries at program start

Global Offset Table (GOT) Protection



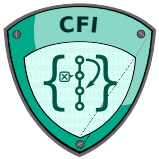
- Observation: only runtime linker populates GOT
  - Any other GOT manipulation is either an accident or an attack
- Idea: Make GOT read-only
- Implementation
  - Linker populates *all* GOT entries at program start → slowdown

Global Offset Table (GOT) Protection

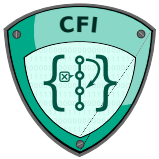


- Observation: only runtime linker populates GOT
  - Any other GOT manipulation is either an accident or an attack
- Idea: Make GOT read-only
- Implementation
  - Linker populates *all* GOT entries at program start → slowdown
  - Compiler flag `-Wl,-z,relro` "relocations read-only"

What are **legitimate targets** for an indirect function call?

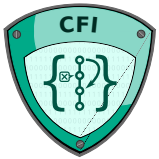


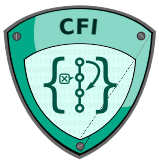
What are **legitimate targets** for an indirect function call? Possible answers:



What are **legitimate targets** for an indirect function call? Possible answers:

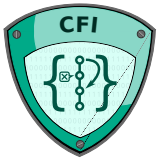
- A1: **any** function





What are **legitimate targets** for an indirect function call? Possible answers:

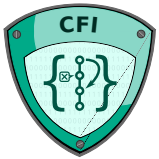
- A1: **any** function
  - Simple but imprecise



What are **legitimate targets** for an indirect function call? Possible answers:

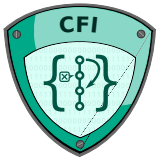
- A1: **any** function
  - Simple but imprecise
  - Attacker can invoke arbitrary functions without violating CFI

Forward-edge CFI



What are **legitimate targets** for an indirect function call? Possible answers:

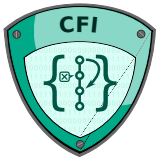
- A1: **any** function
  - Simple but imprecise
  - Attacker can invoke arbitrary functions without violating CFI
- A2: functions with the **same signature** as the function pointer



What are **legitimate targets** for an indirect function call? Possible answers:

- A1: **any** function
  - Simple but imprecise
  - Attacker can invoke arbitrary functions without violating CFI
- A2: functions with the **same signature** as the function pointer
  - Better, prevents some type confusion attacks

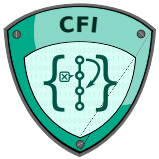
Forward-edge CFI



What are **legitimate targets** for an indirect function call? Possible answers:

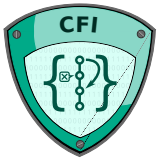
- A1: **any** function
  - Simple but imprecise
  - Attacker can invoke arbitrary functions without violating CFI
- A2: functions with the **same signature** as the function pointer
  - Better, prevents some type confusion attacks
- A3: only functions which **could be assigned** to the function pointer

Forward-edge CFI



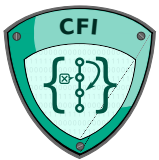
What are **legitimate targets** for an indirect function call? Possible answers:

- A1: **any** function
  - Simple but imprecise
  - Attacker can invoke arbitrary functions without violating CFI
- A2: functions with the **same signature** as the function pointer
  - Better, prevents some type confusion attacks
- A3: only functions which **could be assigned** to the function pointer
  - Even better, still a bit imprecise



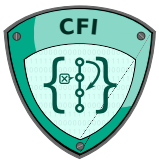
What are **legitimate targets** for an indirect function call? Possible answers:

- A1: **any** function
  - Simple but imprecise
  - Attacker can invoke arbitrary functions without violating CFI
- A2: functions with the **same signature** as the function pointer
  - Better, prevents some type confusion attacks
- A3: only functions which **could be assigned** to the function pointer
  - Even better, still a bit imprecise
- A4: only the function which is **actually assigned** to the function pointer



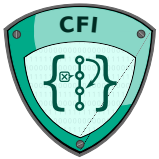
What are **legitimate targets** for an indirect function call? Possible answers:

- A1: **any** function
  - Simple but imprecise
  - Attacker can invoke arbitrary functions without violating CFI
- A2: functions with the **same signature** as the function pointer
  - Better, prevents some type confusion attacks
- A3: only functions which **could be assigned** to the function pointer
  - Even better, still a bit imprecise
- A4: only the function which is **actually assigned** to the function pointer
  - Fully precise, more expensive

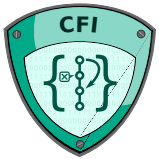


What are **legitimate targets** for an indirect function call? Possible answers:

- A1: **any** function
  - Simple but imprecise
  - Attacker can invoke arbitrary functions without violating CFI
- A2: functions with the **same signature** as the function pointer
  - Better, prevents some type confusion attacks
- A3: only functions which **could be assigned** to the function pointer
  - Even better, still a bit imprecise
- A4: only the function which is **actually assigned** to the function pointer
  - Fully precise, more expensive
  - Code pointer integrity

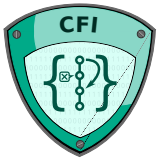


Examples for Hardware/Software CFI mechanisms



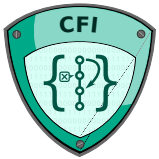
Examples for Hardware/Software CFI mechanisms

- A1: **any** function is valid



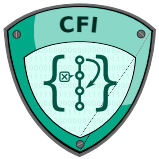
Examples for Hardware/Software CFI mechanisms

- A1: **any** function is valid
  - Hardware CFI: Control-Flow Enforcement Technology (CET)



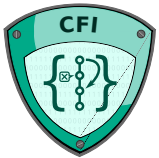
Examples for Hardware/Software CFI mechanisms

- A1: **any** function is valid
  - Hardware CFI: Control-Flow Enforcement Technology (CET)
- A2: functions with the **same signature** are valid



Examples for Hardware/Software CFI mechanisms

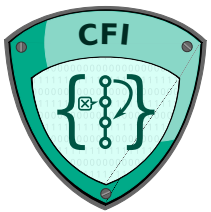
- A1: **any** function is valid
  - Hardware CFI: Control-Flow Enforcement Technology (CET)
- A2: functions with the **same signature** are valid
  - Software CFI: clang `-fsanitize=cfi`



Examples for Hardware/Software CFI mechanisms

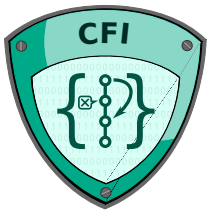
- A1: **any** function is valid
  - Hardware CFI: Control-Flow Enforcement Technology (CET)
- A2: functions with the **same signature** are valid
  - Software CFI: clang `-fsanitize=cfi`
  - Hardware CFI: Arm Pointer Authentication, FineIBT

- Every function entry marked with `endbr64` instruction



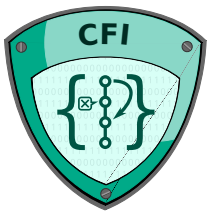
CFI – Control-Flow Enforcement Technology (CET)

- Every function entry marked with `endbr64` instruction
- Call instructions only succeed towards `endbr64` instructions



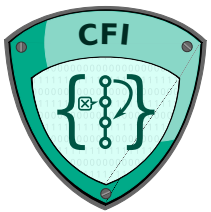
CFI – Control-Flow Enforcement Technology (CET)

- Every function entry marked with `endbr64` instruction
- Call instructions only succeed towards `endbr64` instructions
- Supported by recent Intel and AMD CPUs



CFI – Control-Flow Enforcement Technology (CET)

- Every function entry marked with `endbr64` instruction
- Call instructions only succeed towards `endbr64` instructions
- Supported by recent Intel and AMD CPUs




```
% objdump -d main
0000000000001149 <test>:
    1149:  f3 0f 1e fa    endbr64
    114d:  55             push   %rbp
    114e:  48 89 e5       mov    %rsp,%rbp
    ...

0000000000001160 <test2>:
    1160:  f3 0f 1e fa    endbr64
    1164:  55             push   %rbp
    1165:  48 89 e5       mov    %rsp,%rbp
    ...
```

CFI – Clang I

```cpp
#include <iostream>

class A {
  public: virtual const char* name() { return "A"; };
};
class B {
  public:  const char* name() { return "B"; };
  private: virtual const char* secret() { return "secret"; };
};

int main() {
  A* a = new A();
  std::cout << a->name() << std::endl;
  B* b = new B();
  std::cout << b->name() << std::endl;

  a = (A*)b; // type confusion vulnerability
  std::cout << a->name() << std::endl;
}
```

```
% ./test
A
```

```
% ./test
A
B
```

```
% ./test
A
B
secret
```

CFI – Clang II

```
% ./test
A
B
secret
```

```
% clang++ -flto -fsanitize=cfi -fvisibility=hidden \
  -fno-sanitize-trap=all tc.cpp -o tc
```

CFI – Clang II

```
% ./test
A
B
secret
```

```
% clang++ -flto -fsanitize=cfi -fvisibility=hidden \
  -fno-sanitize-trap=all tc.cpp -o tc
% ./tc
A
```

CFI – Clang II

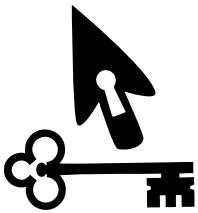
```
% ./test
A
B
secret
```

```
% clang++ -flto -fsanitize=cfi -fvisibility=hidden \
  -fno-sanitize-trap=all tc.cpp -o tc
% ./tc
A
B
```

CFI – Clang II

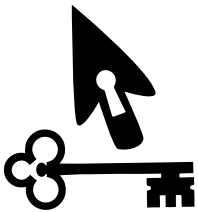
```
% ./test
A
B
secret
```

```
% clang++ -flto -fsanitize=cfi -fvisibility=hidden \
  -fno-sanitize-trap=all tc.cpp -o tc
% ./tc
A
B
tc.cpp:21:9: runtime error: control flow integrity check for
type 'A' failed during cast to unrelated type
(vtable address 0x00000042bd40)
0x00000042bd40: note: vtable is of type 'B'
00 00 00 00  d0 4b 42 00 00 00 00 00  01 1b 03 3b cc 11 00
             ^
```



ARMv8.3 hardware-based pointer authentication

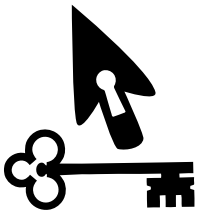
CFI – Pointer Authentication



ARMv8.3 hardware-based pointer authentication

- 👁 Observation: 64-bit architectures do not use all bits

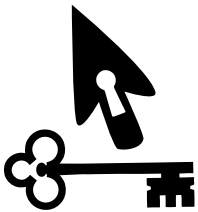
CFI – Pointer Authentication



ARMv8.3 hardware-based pointer authentication

- 👁 Observation: 64-bit architectures do not use all bits
  - E.g. 48-bit virtual address space → 16 bits unused

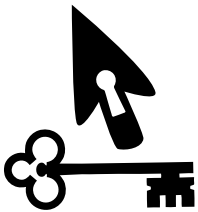
CFI – Pointer Authentication



ARMv8.3 hardware-based pointer authentication

- Observation: 64-bit architectures do not use all bits
  - E.g. 48-bit virtual address space → 16 bits unused
- Idea: repurpose bits for cryptograhpically authenticating pointers

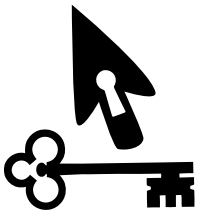
CFI – Pointer Authentication



ARMv8.3 hardware-based pointer authentication

- Observation: 64-bit architectures do not use all bits
  - E.g. 48-bit virtual address space → 16 bits unused
- Idea: repurpose bits for cryptograhpically authenticating pointers
- Implementation

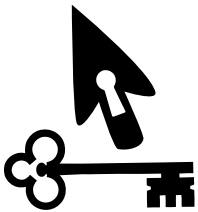
CFI – Pointer Authentication



ARMv8.3 hardware-based pointer authentication

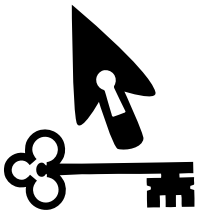
- Observation: 64-bit architectures do not use all bits
  - E.g. 48-bit virtual address space → 16 bits unused
- Idea: repurpose bits for cryptograhpically authenticating pointers
- Implementation
  - Compiler secures a pointer with special `PAC` instructions

CFI – Pointer Authentication



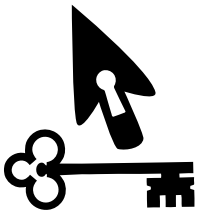
ARMv8.3 hardware-based pointer authentication

- Observation: 64-bit architectures do not use all bits
  - E.g. 48-bit virtual address space → 16 bits unused
- Idea: repurpose bits for cryptograhpically authenticating pointers
- Implementation
  - Compiler secures a pointer with special `PAC` instructions
  - Hardware computes cryptographic MAC



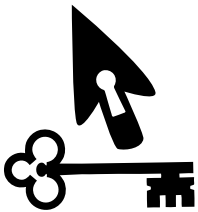
ARMv8.3 hardware-based pointer authentication

- Observation: 64-bit architectures do not use all bits
  - E.g. 48-bit virtual address space → 16 bits unused
- Idea: repurpose bits for cryptograhpically authenticating pointers
- Implementation
  - Compiler secures a pointer with special `PAC` instructions
  - Hardware computes cryptographic MAC
  - Hardware stores parts of the MAC in unused pointer bits



ARMv8.3 hardware-based pointer authentication

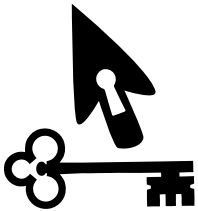
- Observation: 64-bit architectures do not use all bits
  - E.g. 48-bit virtual address space → 16 bits unused
- Idea: repurpose bits for cryptograhpically authenticating pointers
- Implementation
  - Compiler secures a pointer with special `PAC` instructions
  - Hardware computes cryptographic MAC
  - Hardware stores parts of the MAC in unused pointer bits
  - Before dereferencing (calling) a pointer, AUT instruction authenticates

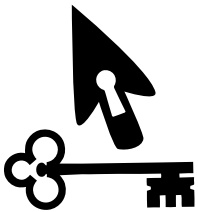


ARMv8.3 hardware-based pointer authentication

- Observation: 64-bit architectures do not use all bits
  - E.g. 48-bit virtual address space → 16 bits unused
- Idea: repurpose bits for cryptograhpically authenticating pointers
- Implementation
  - Compiler secures a pointer with special `PAC` instructions
  - Hardware computes cryptographic MAC
  - Hardware stores parts of the MAC in unused pointer bits
  - Before dereferencing (calling) a pointer, AUT instruction authenticates
  - Hardware invalidates pointer or faults on authentication failure

MAC algorithm has three inputs

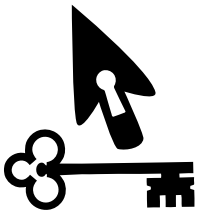




MAC algorithm has three inputs

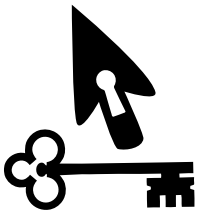
- Pointer value

CFI – Pointer Authentication



MAC algorithm has three inputs

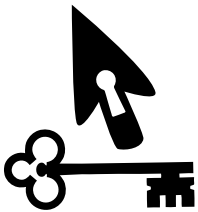
- Pointer value
- Secret (process-dependent) key



MAC algorithm has three inputs

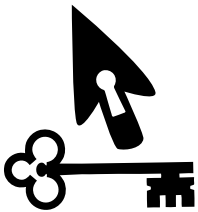
- Pointer value
- Secret (process-dependent) key
- User-defined *context*

CFI – Pointer Authentication



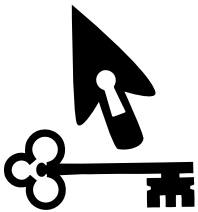
MAC algorithm has three inputs

- Pointer value
- Secret (process-dependent) key
- User-defined *context*
- *Context* can be used to distinguish



MAC algorithm has three inputs

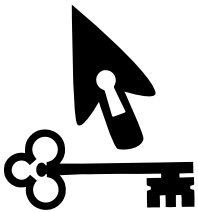
- Pointer value
- Secret (process-dependent) key
- User-defined *context*
- *Context* can be used to distinguish
  - Function signatures



MAC algorithm has three inputs

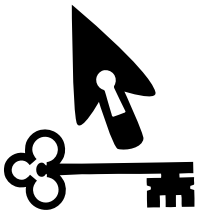
- Pointer value
- Secret (process-dependent) key
- User-defined *context*
- *Context* can be used to distinguish
  - Function signatures
  - Types of data pointers

CFI – Pointer Authentication



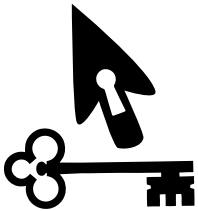
MAC algorithm has three inputs

- Pointer value
- Secret (process-dependent) key
- User-defined *context*
- *Context* can be used to distinguish
  - Function signatures
  - Types of data pointers
  - Stack frames

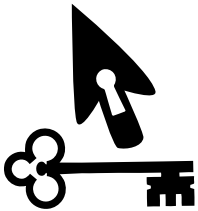


MAC algorithm has three inputs

- Pointer value
- Secret (process-dependent) key
- User-defined *context*
- *Context* can be used to distinguish
  - Function signatures
  - Types of data pointers
  - Stack frames
  - ...

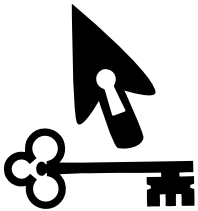


- Properties

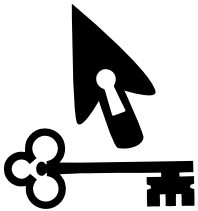


- Properties
  - Cryptographically-enforced CFI

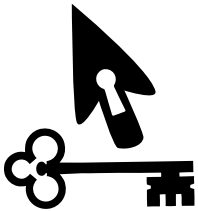
CFI – Pointer Authentication



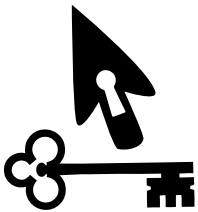
- Properties
  - Cryptographically-enforced CFI
  - Effectiveness depends on additional *context* input



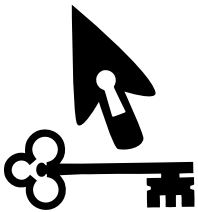
- Properties
  - Cryptographically-enforced CFI
  - Effectiveness depends on additional *context* input
    - Precision: if *context* is zero, attacker can exchange all authenticated pointers (Similar to Intel CET)



- Properties
  - Cryptographically-enforced CFI
  - Effectiveness depends on additional *context* input
    - Precision: if *context* is zero, attacker can exchange all authenticated pointers (Similar to Intel CET)
    - Security: If attacker can control *context* → forge arbitrary pointers

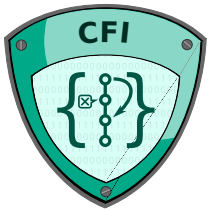


- Properties
  - Cryptographically-enforced CFI
  - Effectiveness depends on additional *context* input
    - Precision: if *context* is zero, attacker can exchange all authenticated pointers (Similar to Intel CET)
    - Security: If attacker can control *context* → forge arbitrary pointers
  - Authentication code (MAC) is truncated to upper pointer bits



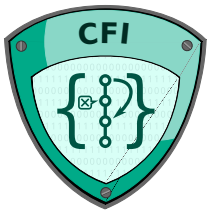
- Properties
  - Cryptographically-enforced CFI
  - Effectiveness depends on additional *context* input
    - Precision: if *context* is zero, attacker can exchange all authenticated pointers (Similar to Intel CET)
    - Security: If attacker can control *context* → forge arbitrary pointers
  - Authentication code (MAC) is truncated to upper pointer bits
    - Imprecision, allows brute-force/collision attacks

Control Flow Integrity (CFI) Summary



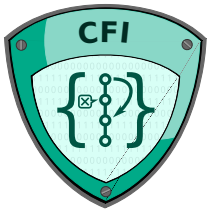
- Properties

Control Flow Integrity (CFI) Summary



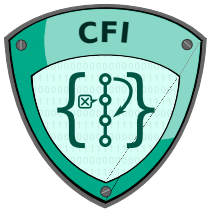
- Properties
  - CFI: attacker cannot escape control flow graph (CFG)

Control Flow Integrity (CFI) Summary



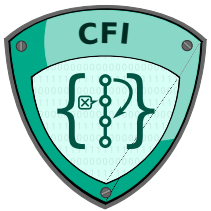
- Properties
  - CFI: attacker cannot escape control flow graph (CFG)
    - Defeats ROP, JOP, ...

Control Flow Integrity (CFI) Summary



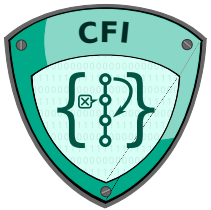
- Properties
  - CFI: attacker cannot escape control flow graph (CFG)
    - Defeats ROP, JOP, ...
    - Still allows more or less code reuse within the CFG

Control Flow Integrity (CFI) Summary



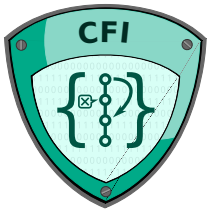
- Properties
  - CFI: attacker cannot escape control flow graph (CFG)
    - Defeats ROP, JOP, ...
    - Still allows more or less code reuse within the CFG
    - Depending on precision of forward edges, attacker can substitute valid pointers

Control Flow Integrity (CFI) Summary



- Properties
  - CFI: attacker cannot escape control flow graph (CFG)
    - Defeats ROP, JOP, ...
    - Still allows more or less code reuse within the CFG
    - Depending on precision of forward edges, attacker can substitute valid pointers
  - CFI does not prevent data-only attacks

Control Flow Integrity (CFI) Summary



- Properties
  - CFI: attacker cannot escape control flow graph (CFG)
    - Defeats ROP, JOP, ...
    - Still allows more or less code reuse within the CFG
    - Depending on precision of forward edges, attacker can substitute valid pointers
  - CFI does not prevent data-only attacks
    - E.g., is_admin flag, loop counters, syscall arguments?

Memory Safety

- Observation: Most attacks due to a memory safety vulnerability

Memory Safety

- Observation: Most attacks due to a memory safety vulnerability
- Idea: Prevent exploitation of memory safety vulnerability

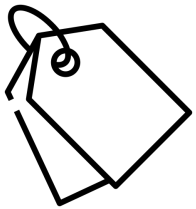
Memory Safety

- Observation: Most attacks due to a memory safety vulnerability
- Idea: Prevent exploitation of memory safety vulnerability
  - Preventing invalid memory access

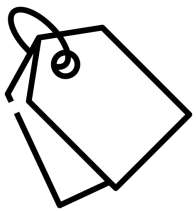
Memory Safety

- Observation: Most attacks due to a memory safety vulnerability
- Idea: Prevent exploitation of memory safety vulnerability
  - Preventing invalid memory access
  - ARM Memory Tagging Extention (MTE)

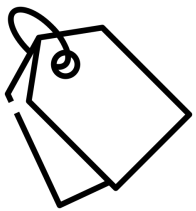
Memory Tagging Extention



- On allocation

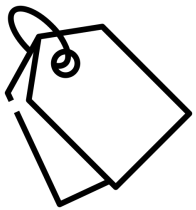


- On allocation
  - e.g., `char *ptr = malloc(8);`



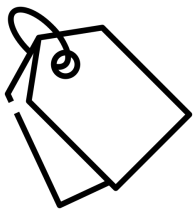
- On allocation
  - e.g., `char *ptr = malloc(8);`
  - Tag memory object

Memory Tagging I

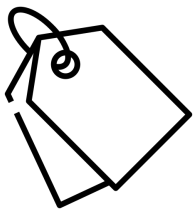


- On allocation
  - e.g., `char *ptr = malloc(8);`
  - Tag memory object
  - Store tag in pointer

Memory Tagging I

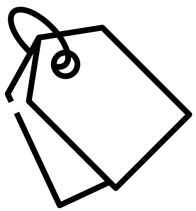


- On allocation
  - e.g., `char *ptr = malloc(8);`
  - Tag memory object
  - Store tag in pointer
- On access

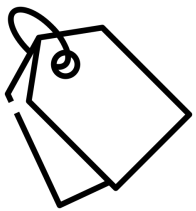


- On allocation
  - e.g., `char *ptr = malloc(8);`
  - Tag memory object
  - Store tag in pointer
- On access
  - Check whether memory object is tagged with the stored tag of the pointer

Memory Tagging I

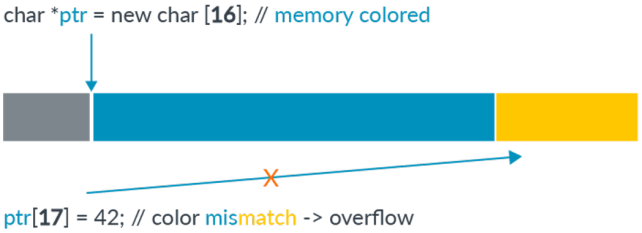


- On allocation
  - e.g., `char *ptr = malloc(8);`
  - Tag memory object
  - Store tag in pointer
- On access
  - Check whether memory object is tagged with the stored tag of the pointer
  - If correct → Access



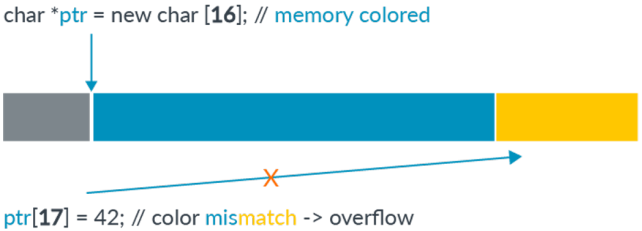
- On allocation
  - e.g., `char *ptr = malloc(8);`
  - Tag memory object
  - Store tag in pointer
- On access
  - Check whether memory object is tagged with the stored tag of the pointer
  - If correct → Access
  - If not correct → Fault

Memory Tagging II



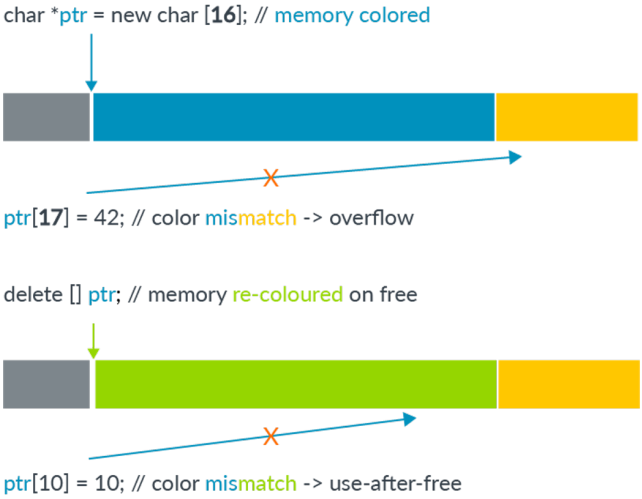
- Security claims

Memory Tagging II



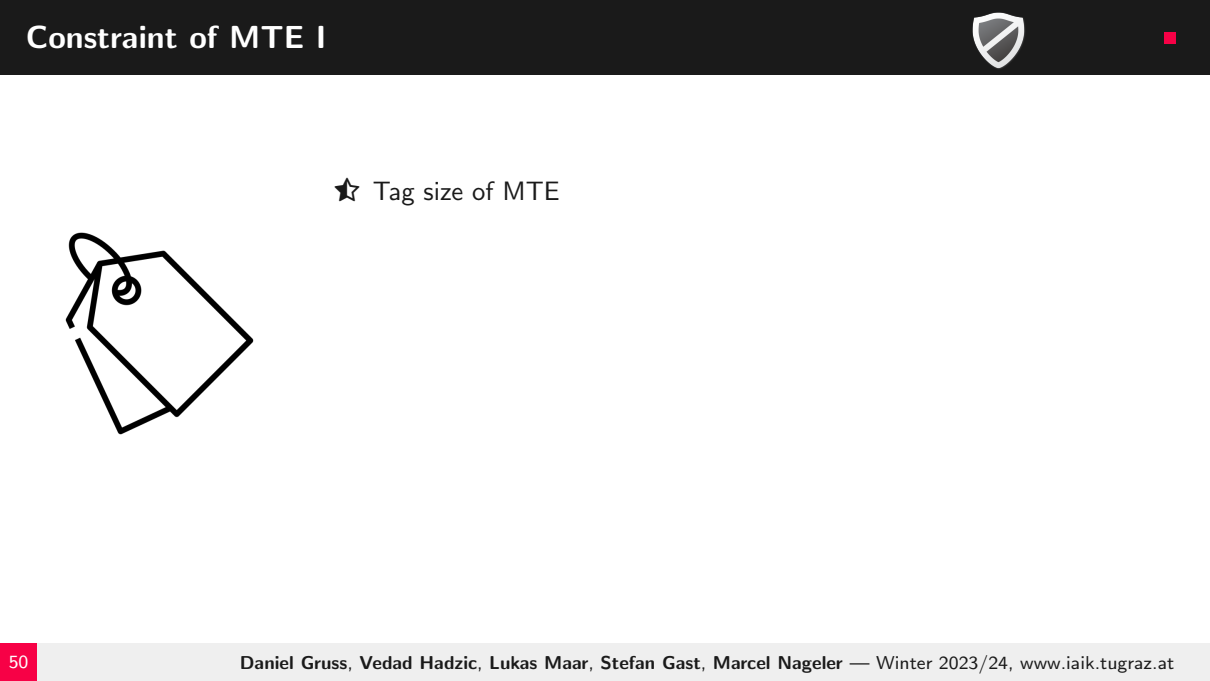
- Security claims
  - Prevents spatial memory violations

Memory Tagging II



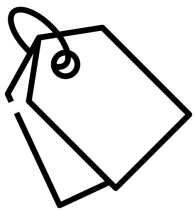
- Security claims
  - Prevents spatial memory violations
  - Prevents temporal memory violations

Constraint of MTE I



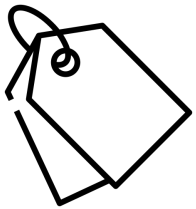
- Tag size of MTE

Constraint of MTE I



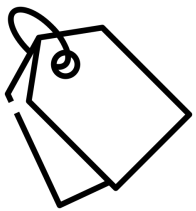
- Tag size of MTE
  - 4 bits

Constraint of MTE I



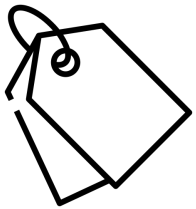
- Tag size of MTE
  - 4 bits
  - 16 distinct tags

Constraint of MTE I



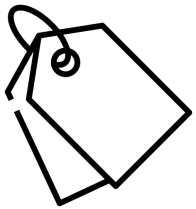
- Tag size of MTE
  - 4 bits
  - 16 distinct tags
- Tag storage

Constraint of MTE I



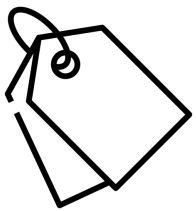
- Tag size of MTE
  - 4 bits
  - 16 distinct tags
- Tag storage
  - Pointer

Constraint of MTE I



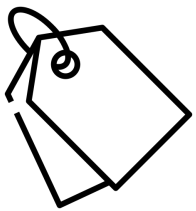
- Tag size of MTE
  - 4 bits
  - 16 distinct tags
- Tag storage
  - Pointer
    - Only 48 bits used of virtual address

Constraint of MTE I



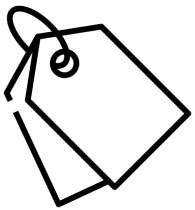
- Tag size of MTE
  - 4 bits
  - 16 distinct tags
- Tag storage
  - Pointer
    - Only 48 bits used of virtual address
    - Store 4 bit tag in unused bits

Constraint of MTE I



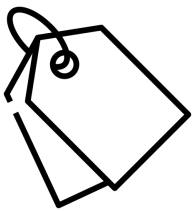
- Tag size of MTE
  - 4 bits
  - 16 distinct tags
- Tag storage
  - Pointer
    - Only 48 bits used of virtual address
    - Store 4 bit tag in unused bits
  - Memory object

Constraint of MTE I



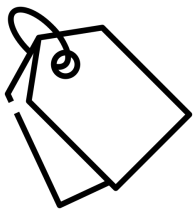
- Tag size of MTE
  - 4 bits
  - 16 distinct tags
- Tag storage
  - Pointer
    - Only 48 bits used of virtual address
    - Store 4 bit tag in unused bits
  - Memory object
    - In memory

Constraint of MTE I



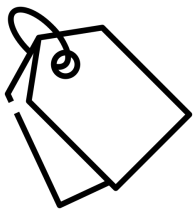
- Tag size of MTE
  - 4 bits
  - 16 distinct tags
- Tag storage
  - Pointer
    - Only 48 bits used of virtual address
    - Store 4 bit tag in unused bits
  - Memory object
    - In memory
    - 3.125 % memory overhead with 4 bit tag size

Constraint of MTE I



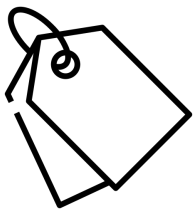
- Tag size of MTE
  - 4 bits
  - 16 distinct tags
- Tag storage
  - Pointer
    - Only 48 bits used of virtual address
    - Store 4 bit tag in unused bits
  - Memory object
    - In memory
    - 3.125 % memory overhead with 4 bit tag size
    - Two memory lookups for dereferencing a pointer

Constraint of MTE II



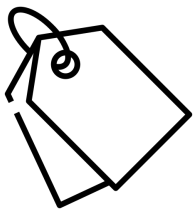
- Given 16 tags, how likely is it to detect an arbitrary memory access or a use-after-free?

Constraint of MTE II



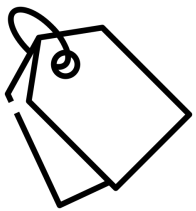
- Given 16 tags, how likely is it to detect an arbitrary memory access or a use-after-free?
  - On random re-coloring

Constraint of MTE II



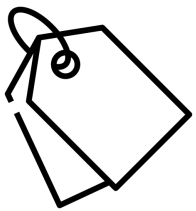
- Given 16 tags, how likely is it to detect an arbitrary memory access or a use-after-free?
  - On random re-coloring
  - $\left(1 - \frac{1}{2^4}\right) \cdot 100 = 93.75\,\%$

Constraint of MTE II



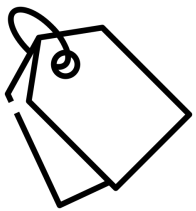
- Given 16 tags, how likely is it to detect an arbitrary memory access or a use-after-free?
  - On random re-coloring
  - $\left(1 - \frac{1}{2^4}\right) \cdot 100 = 93.75\,\%$
- Increase tag size to 8 bits?

Constraint of MTE II



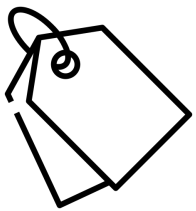
- Given 16 tags, how likely is it to detect an arbitrary memory access or a use-after-free?
  - On random re-coloring
  - $\left(1 - \frac{1}{2^4}\right) \cdot 100 = 93.75\,\%$
- Increase tag size to 8 bits?
  - Detection propability increases to $\left(1 - \frac{1}{2^8}\right) \cdot 100 \approx 99.6\,\%$

Constraint of MTE II



- Given 16 tags, how likely is it to detect an arbitrary memory access or a use-after-free?
  - On random re-coloring
  - $\left(1 - \frac{1}{2^4}\right) \cdot 100 = 93.75\,\%$
- Increase tag size to 8 bits?
  - Detection propability increases to $\left(1 - \frac{1}{2^8}\right) \cdot 100 \approx 99.6\,\%$
  - Memory overhead increases to 6.25 %

Constraint of MTE II



- Given 16 tags, how likely is it to detect an arbitrary memory access or a use-after-free?
  - On random re-coloring
  - $\left(1 - \frac{1}{2^4}\right) \cdot 100 = 93.75\,\%$
- Increase tag size to 8 bits?
  - Detection propability increases to $\left(1 - \frac{1}{2^8}\right) \cdot 100 \approx 99.6\,\%$
  - Memory overhead increases to 6.25 %
  - Currently only 4 bit supported

Defense In Depth

Attacker's perspective

- Vulnerability discovery ☑
- Exploitation ☑
- Privilege elevation ☑

Defender's perspective

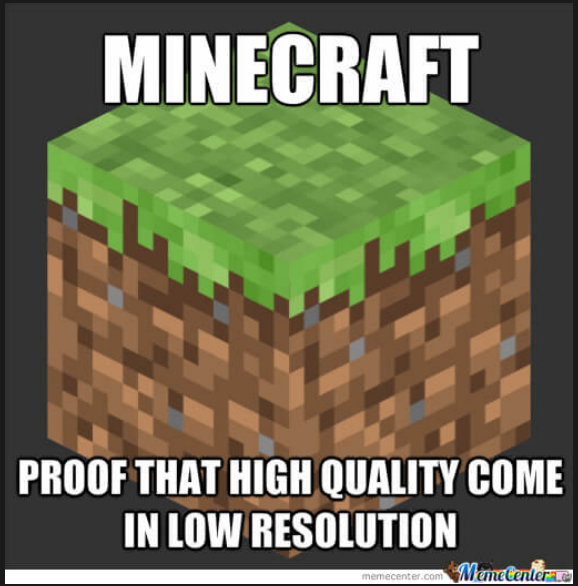
- Vulnerability prevention ☑
- Exploit prevention ☑
- Privilege minimization (today)

Privilege Minimization

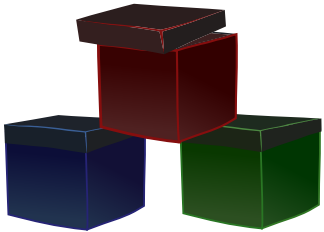
How to minimize the impact of Arbitrary Code Execution

Think inside boxes ...

Think inside boxes ...

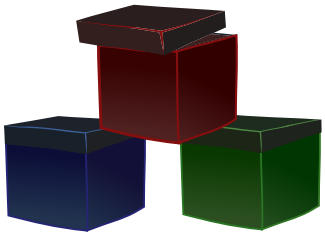


Privilege Minimization Overview



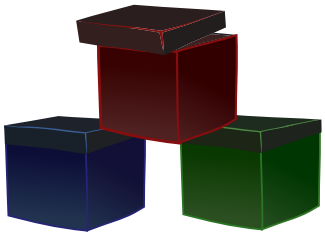
- Everything is a box
  - "Principle of least privileges"

Privilege Minimization Overview



- Everything is a box
  - "Principle of least privileges"
- Compartmentalization
  - Break large boxes into smaller boxes
  - Virtual machines, processes, libraries, functions ...
  - Mostly manual effort

Privilege Minimization Overview

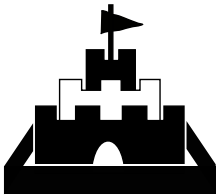


- Everything is a box
  - "Principle of least privileges"
- Compartmentalization
  - Break large boxes into smaller boxes
  - Virtual machines, processes, libraries, functions ...
  - Mostly manual effort
- Isolation
  - Isolate boxes from each other
  - Safeguard all interfaces
    - File permissions, network firewall ... system calls

In-process Sandboxing

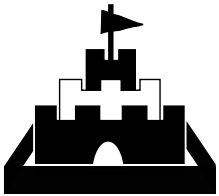
In-process Sandboxing

- Goal: confine parts of an application



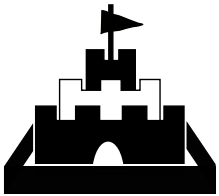
- Goal: confine parts of an application
  - E.g., dangerous plugins, libraries, user-provided code ...

In-process Sandboxing



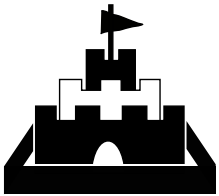
- Goal: confine parts of an application
  - E.g., dangerous plugins, libraries, user-provided code ...
- Dangerous code is not executed natively but interpreted

In-process Sandboxing



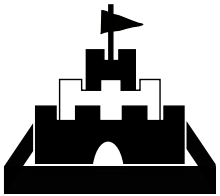
- Goal: confine parts of an application
  - E.g., dangerous plugins, libraries, user-provided code ...
- Dangerous code is not executed natively but interpreted
  - E.g., JavaScript, WebAssembly, ...

In-process Sandboxing



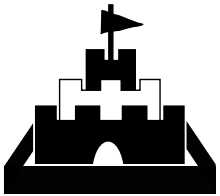
- Goal: confine parts of an application
  - E.g., dangerous plugins, libraries, user-provided code ...
- Dangerous code is not executed natively but interpreted
  - E.g., JavaScript, WebAssembly, ...
- Software-enforced:

In-process Sandboxing



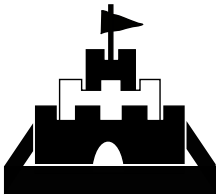
- Goal: confine parts of an application
  - E.g., dangerous plugins, libraries, user-provided code ...
- Dangerous code is not executed natively but interpreted
  - E.g., JavaScript, WebAssembly, ...
- Software-enforced:
  - Software-Fault Isolation (SFI)

In-process Sandboxing



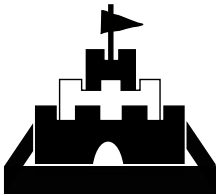
- Goal: confine parts of an application
  - E.g., dangerous plugins, libraries, user-provided code ...
- Dangerous code is not executed natively but interpreted
  - E.g., JavaScript, WebAssembly, ...
- Software-enforced:
  - Software-Fault Isolation (SFI)
  - Masks access to restrict memory accesses

In-process Sandboxing



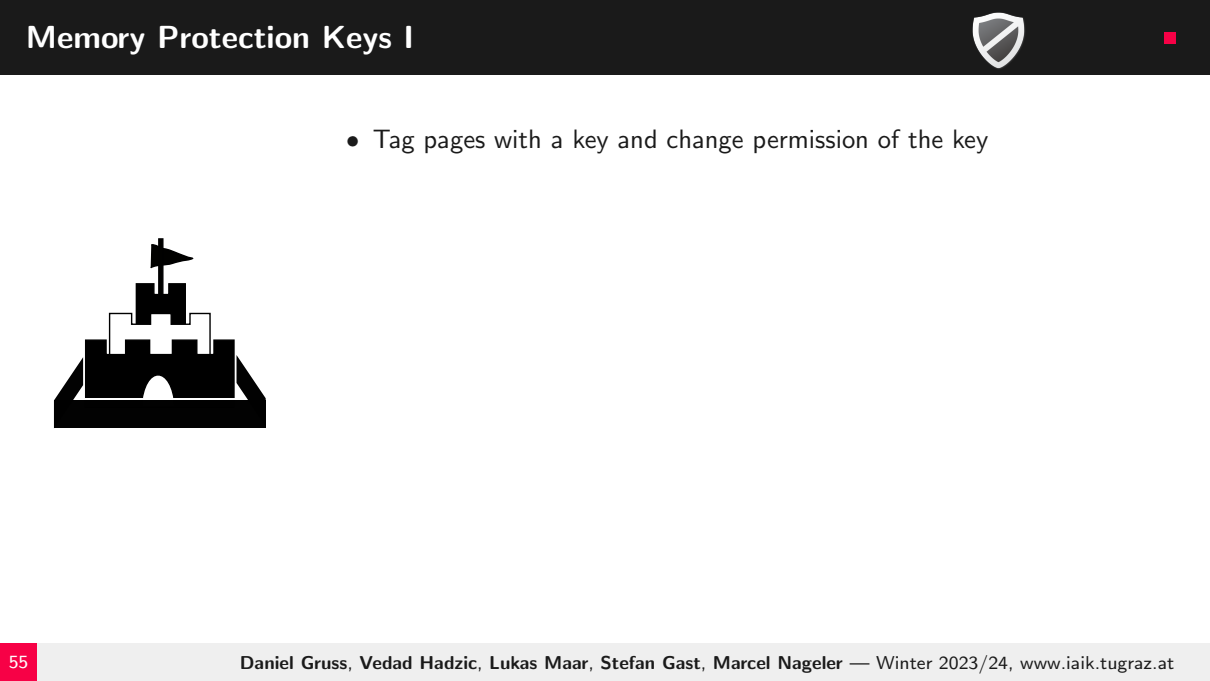
- Goal: confine parts of an application
  - E.g., dangerous plugins, libraries, user-provided code ...
- Dangerous code is not executed natively but interpreted
  - E.g., JavaScript, WebAssembly, ...
- Software-enforced:
  - Software-Fault Isolation (SFI)
  - Masks access to restrict memory accesses
- Hardware-enforced:

In-process Sandboxing



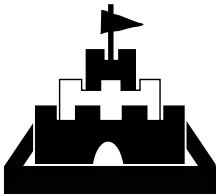
- Goal: confine parts of an application
  - E.g., dangerous plugins, libraries, user-provided code ...
- Dangerous code is not executed natively but interpreted
  - E.g., JavaScript, WebAssembly, ...
- Software-enforced:
  - Software-Fault Isolation (SFI)
  - Masks access to restrict memory accesses
- Hardware-enforced:
  - Memory Protection Keys (MPK)

Memory Protection Keys I



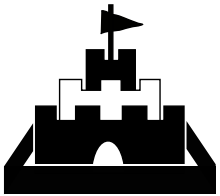
- Tag pages with a key and change permission of the key

Memory Protection Keys I



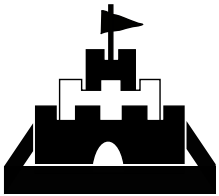
- Tag pages with a key and change permission of the key
- Change access permissions of a key $\rightarrow$ change permission the tagged pages

Memory Protection Keys I



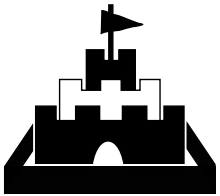
- Tag pages with a key and change permission of the key
- Change access permissions of a key $\rightarrow$ change permission the tagged pages
- Tagged key stored in page-table

Memory Protection Keys I



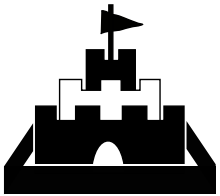
- Tag pages with a key and change permission of the key
- Change access permissions of a key $\rightarrow$ change permission the tagged pages
- Tagged key stored in page-table
- Keys permissions are stored in dedicated register

Memory Protection Keys I



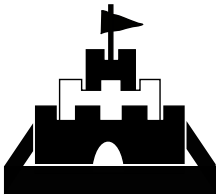
- Tag pages with a key and change permission of the key
- Change access permissions of a key $\rightarrow$ change permission the tagged pages
- Tagged key stored in page-table
- Keys permissions are stored in dedicated register
- Allow fine-grained permission setting, *i.e.*, read/write-access, of tagged page

Memory Protection Keys I



- Tag pages with a key and change permission of the key
- Change access permissions of a key → change permission the tagged pages
- Tagged key stored in page-table
- Keys permissions are stored in dedicated register
- Allow fine-grained permission setting, *i.e.*, read/write-access, of tagged page
- Hardware implementations:
  - Intel Protection Keys for Userspace (PKU) and Protection Keys for Supervisor (PKS)
  - ARM Domain Access Control (DAC)

Memory Protection Keys I



- Tag pages with a key and change permission of the key
- Change access permissions of a key → change permission the tagged pages
- Tagged key stored in page-table
- Keys permissions are stored in dedicated register
- Allow fine-grained permission setting, *i.e.*, read/write-access, of tagged page
- Hardware implementations:
  - Intel Protection Keys for Userspace (PKU) and Protection Keys for Supervisor (PKS)
  - ARM Domain Access Control (DAC)
- Only 4 bit key → 16 distinct access domains

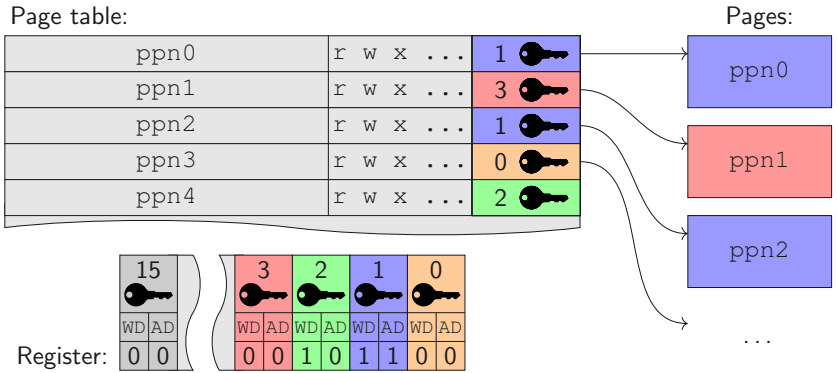


Figure 1: Working prinicple of MPK, where `AD` and `WD` stands for access and write disable, respectively. Pages tagged with key 0 are write- and access-permitted, while pages tagged with key 1 are write- and access-prohibited, and pages tagged with key 2 are only write-prohibited.

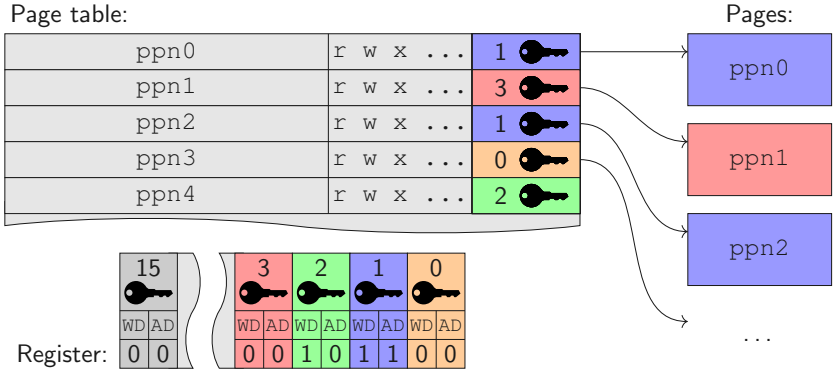


Figure 1: Working prinicple of MPK, where `AD` and `WD` stands for access and write disable, respectively. Pages tagged with key 0 are write- and access-permitted, while pages tagged with key 1 are write- and access-prohibited, and pages tagged with key 2 are only write-prohibited. Advantages over `mprotect`: No TLB flush and page-table walk on permission change → faster.

Process Sandboxing


SECCOMP

- Observation: Most programs do not need most system calls

- Observation: Most programs do not need most system calls
  - E.g., fork, exec, prctl ...

- Observation: Most programs do not need most system calls
  - E.g., fork, exec, prctl ...
- Idea: block unnecessary system calls

- Observation: Most programs do not need most system calls
  - E.g., fork, exec, prctl ...
- Idea: block unnecessary system calls
- Implementation

- Observation: Most programs do not need most system calls
  - E.g., fork, exec, prctl ...
- Idea: block unnecessary system calls
- Implementation
  - Program installs seccomp filters on startup

Seccomp

- Observation: Most programs do not need most system calls
  - E.g., fork, exec, prctl ...
- Idea: block unnecessary system calls
- Implementation
  - Program installs seccomp filters on startup
  - Seccomp supports small *Berkeley Packet Filter (BPF) programs*

- Observation: Most programs do not need most system calls
  - E.g., fork, exec, prctl ...
- Idea: block unnecessary system calls
- Implementation
  - Program installs seccomp filters on startup
  - Seccomp supports small *Berkeley Packet Filter (BPF) programs*
  - Kernel does the filtering (e.g., executes the BPF program) on every system call

- Observation: Most programs do not need most system calls
  - E.g., fork, exec, prctl ...
- Idea: block unnecessary system calls
- Implementation
  - Program installs seccomp filters on startup
  - Seccomp supports small *Berkeley Packet Filter (BPF) programs*
  - Kernel does the filtering (e.g., executes the BPF program) on every system call
  - On a filter violation: deny syscall, send signal, kill program ...

Seccomp Example I

```c
#include <stdio.h>          /* printf */
#include <sys/prctl.h>      /* prctl */
#include <linux/seccomp.h> /* seccomp's constants */
#include <unistd.h>         /* dup2: just for test */

int main() {
  printf("step 1: unrestricted\n");
  prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT); // Enable filtering
  printf("step 2: only 'read', 'write', '_exit' and 'sigreturn' syscalls\n");
  dup2(1, 2); // redirect stderr to stdout
  printf("step 3: !! YOU SHOULD NOT SEE ME !!\n");
  return 0;
}
```

https://blog.yadutaf.fr/2014/05/29/introduction-to-seccomp-bpf-linux-syscall-filter/

Seccomp Example II

```
dgruss@t460sdg ~ % gcc seccomp.c
dgruss@t460sdg ~ % ./a.out
step 1: unrestricted
step 2: only 'read', 'write', '_exit' and 'sigreturn' syscalls
[1]    19622 killed     ./a.out
137 dgruss@t460sdg ~ %
```

Seccomp Example III

```c
int main() {
  printf("step 1: init\n");
  prctl(PR_SET_NO_NEW_PRIVS, 1);
  prctl(PR_SET_DUMPABLE, 0);      // ptrace on this process / childs is not allowed
  scmp_filter_ctx ctx;
  ctx = seccomp_init(SCMP_ACT_KILL);                          // Denylist everything
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(rt_sigreturn), 0);   // Allowlist
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit), 0);           // Allowlist
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit_group), 0);     // Allowlist
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(read), 0);           // Allowlist
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 0);          // Allowlist
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(dup2), 2,            // Allowlist
                   SCMP_A0(SCMP_CMP_EQ, 1), SCMP_A1(SCMP_CMP_EQ, 2)); // Allowlist
  seccomp_load(ctx);
  printf("step 2: only 'write' and dup2(1, 2) syscalls\n");
  dup2(1, 2);    // redirect stderr to stdout
  printf("step 3: stderr redirected to stdout\n");
  dup2(2, 42);   // redirect stderr to stdout
}
```

Seccomp Example IV

```
dgruss@t460sdg ~ % gcc seccomp.c -lseccomp && ./a.out
step 1: init
step 2: only 'write' and dup2(1, 2) syscalls
step 3: stderr redirected to stdout
[1]    23312 invalid system call  ./a.out
159 dgruss@t460sdg ~ %
```

Challenge #7 - Secwrap

Write a secure wrapper binary

- Usage: `./secwrap <command>`
- The wrapper shall start the program specified by `<command>`
- Anything `<command>` does may not be allowed to create new processes!

→ Very convenient to use :)

- Upload your wrapper binary at `https://challenges.sasectf.student.iaik.tugraz.at/secwrap/index.php`
- If it is correct, you will get the flag
- Test system is Debian 11.8, kernel 5.10.0-26

- Sandbox process runs dangerous code

- Sandbox process runs dangerous code
- Monitor process interacts with sandbox via IPC

Simple sandboxing with Seccomp

- Sandbox process runs dangerous code
- Monitor process interacts with sandbox via IPC
  - Minimal filter: only allow required IPC system calls

- Sandbox process runs dangerous code
- Monitor process interacts with sandbox via IPC
  - Minimal filter: only allow required IPC system calls
- Example: Google *sandbox2*
  `https://developers.google.com/sandboxed-api/`

Seccomp Summary

- Properties

- Properties
  - Protect system call interface

Seccomp Summary

- Properties
  - Protect system call interface
  - Filters can only be specialized but not tightened

Seccomp Summary

- Properties
  - Protect system call interface
  - Filters can only be specialized but not tightened
    - Attacker cannot manipulate/unload existing filters

- Properties
  - Protect system call interface
  - Filters can only be specialized but not tightened
    - Attacker cannot manipulate/unload existing filters
  - Filter: simple arithmetic operations on system call arguments

Properties

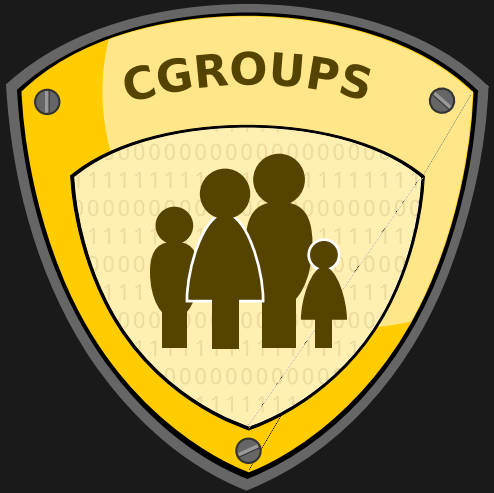
- Protect system call interface
- Filters can only be specialized but not tightened
  - Attacker cannot manipulate/unload existing filters
- Filter: simple arithmetic operations on system call arguments
  - Enhanced filtering is impossible

Seccomp Summary

- Properties
  - Protect system call interface
  - Filters can only be specialized but not tightened
    - Attacker cannot manipulate/unload existing filters
  - Filter: simple arithmetic operations on system call arguments
    - Enhanced filtering is impossible
    - E.g., checking for strings, sanitizing paths, dereferencing pointers

- Properties
  - Protect system call interface
  - Filters can only be specialized but not tightened
    - Attacker cannot manipulate/unload existing filters
  - Filter: simple arithmetic operations on system call arguments
    - Enhanced filtering is impossible
    - E.g., checking for strings, sanitizing paths, dereferencing pointers
- How do we know which system calls are needed by libc functions such as `pthread_create` ? Implementation defined!

- Properties
  - Protect system call interface
  - Filters can only be specialized but not tightened
    - Attacker cannot manipulate/unload existing filters
  - Filter: simple arithmetic operations on system call arguments
    - Enhanced filtering is impossible
    - E.g., checking for strings, sanitizing paths, dereferencing pointers
- How do we know which system calls are needed by libc functions such as `pthread_create` ? Implementation defined!
- How can we virtualize resources?


CGROUPS

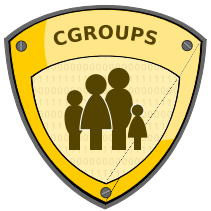
Control groups (cgroups)

- Idea: Manage resource usage of a group of processes (and all its children)

Control groups (cgroups)

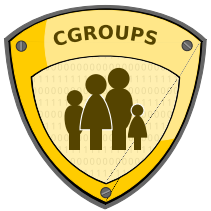
- Idea: Manage resource usage of a group of processes (and all its children)
  - Memory, CPU time, networking, disk I/O ...

Control groups (cgroups)



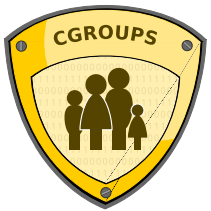
- Idea: Manage resource usage of a group of processes (and all its children)
  - Memory, CPU time, networking, disk I/O ...
  - Set limits / priorities

Control groups (cgroups)



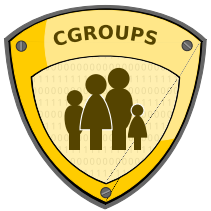
- Idea: Manage resource usage of a group of processes (and all its children)
  - Memory, CPU time, networking, disk I/O ...
  - Set limits / priorities
- Properties

Control groups (cgroups)



- Idea: Manage resource usage of a group of processes (and all its children)
  - Memory, CPU time, networking, disk I/O ...
  - Set limits / priorities
- Properties
  - Can prevent some Denial-of-Service (DoS) attacks

Control groups (cgroups)



- Idea: Manage resource usage of a group of processes (and all its children)
  - Memory, CPU time, networking, disk I/O ...
  - Set limits / priorities
- Properties
  - Can prevent some Denial-of-Service (DoS) attacks
  - Cannot prevent privilege escalation


NAMESPACES

Namespaces

- Idea: Namespace hides (virtualizes) resources from processes

Namespaces

- Idea: Namespace hides (virtualizes) resources from processes
  - Various namespaces: `mnt`, `pid`, `net`, `ipc`, `uts` (hostname), `user`

Namespaces

- Idea: Namespace hides (virtualizes) resources from processes
  - Various namespaces: `mnt`, `pid`, `net`, `ipc`, `uts` (hostname), `user`
  - How? Namespace translates resource identifiers

Namespaces

- Idea: Namespace hides (virtualizes) resources from processes
  - Various namespaces: `mnt`, `pid`, `net`, `ipc`, `uts` (hostname), `user`
  - How? Namespace translates resource identifiers
- Examples:

Namespaces

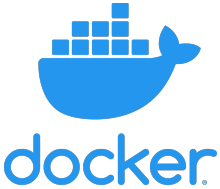
- Idea: Namespace hides (virtualizes) resources from processes
  - Various namespaces: `mnt`, `pid`, `net`, `ipc`, `uts` (hostname), `user`
  - How? Namespace translates resource identifiers
- Examples:
  - Inside namespace: `uid=0(root)`, `path=/f.txt`

Namespaces

- Idea: Namespace hides (virtualizes) resources from processes
  - Various namespaces: `mnt`, `pid`, `net`, `ipc`, `uts` (hostname), `user`
  - How? Namespace translates resource identifiers
- Examples:
  - Inside namespace: `uid=0(root)`, `path=/f.txt`
  - Outside namepsace: `uid=1000(ssd)`, `path=/home/ssd/f.txt`

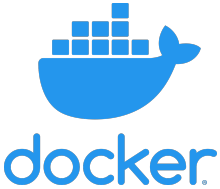
Docker Containers

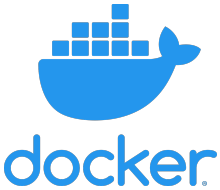
Idea: combine 'em all: Docker containers



Docker Containers

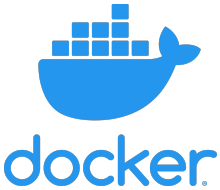
- Idea: combine 'em all: Docker containers
  - See also Linux Containers (LXC)





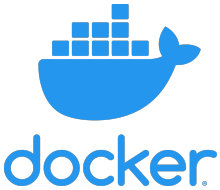
- Idea: combine 'em all: Docker containers
  - See also Linux Containers (LXC)
- Docker automatically

Docker Containers

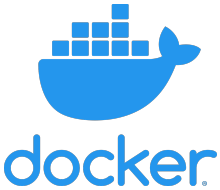


- Idea: combine 'em all: Docker containers
  - See also Linux Containers (LXC)
- Docker automatically
  - creates namespaces and cgroups

Docker Containers

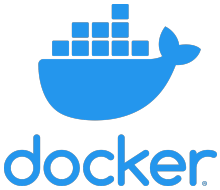


- Idea: combine 'em all: Docker containers
  - See also Linux Containers (LXC)
- Docker automatically
  - creates namespaces and cgroups
  - configures seccomp



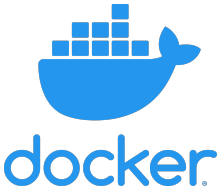
- Idea: combine 'em all: Docker containers
  - See also Linux Containers (LXC)
- Docker automatically
  - creates namespaces and cgroups
  - configures seccomp
- Properties

Docker Containers

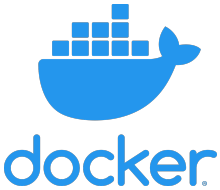


- Idea: combine 'em all: Docker containers
  - See also Linux Containers (LXC)
- Docker automatically
  - creates namespaces and cgroups
  - configures seccomp
- Properties
  - Fast

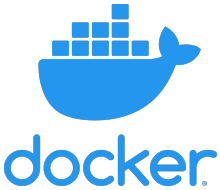
Docker Containers



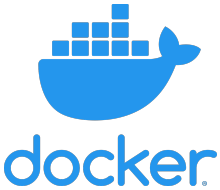
- Idea: combine 'em all: Docker containers
  - See also Linux Containers (LXC)
- Docker automatically
  - creates namespaces and cgroups
  - configures seccomp
- Properties
  - Fast
  - Security depends on proper configuration



- Idea: combine 'em all: Docker containers
  - See also Linux Containers (LXC)
- Docker automatically
  - creates namespaces and cgroups
  - configures seccomp
- Properties
  - Fast
  - Security depends on proper configuration
  - Kernel is shared between containers and host



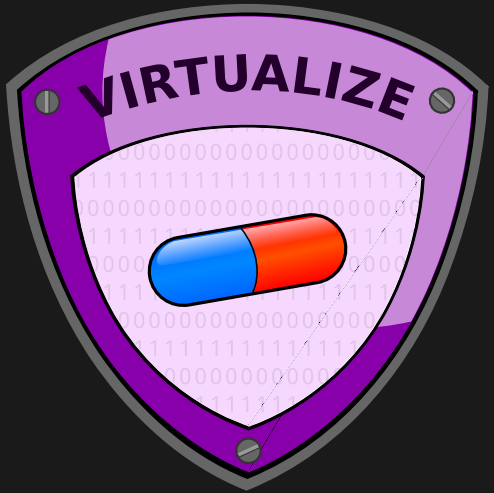
- Idea: combine 'em all: Docker containers
  - See also Linux Containers (LXC)
- Docker automatically
  - creates namespaces and cgroups
  - configures seccomp
- Properties
  - Fast
  - Security depends on proper configuration
  - Kernel is shared between containers and host
- What if one container compromises the host kernel?



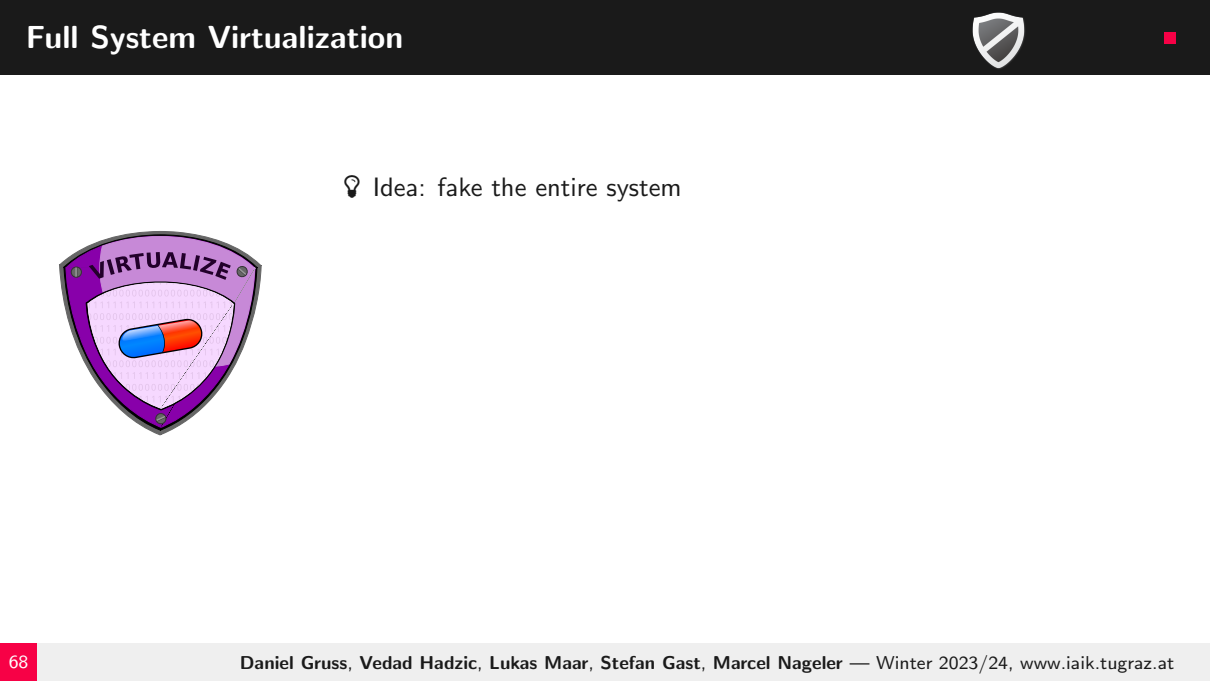
- Idea: combine 'em all: Docker containers
  - See also Linux Containers (LXC)
- Docker automatically
  - creates namespaces and cgroups
  - configures seccomp
- Properties
  - Fast
  - Security depends on proper configuration
  - Kernel is shared between containers and host
- What if one container compromises the host kernel?

  GVisor/Kata to harden Docker containers

Virtualization


VIRTUALIZE

Full System Virtualization



- Idea: fake the entire system

Full System Virtualization

- Idea: fake the entire system
  - Virtualization of hardware resources (memory, CPU, peripherals ...)

Full System Virtualization

- Idea: fake the entire system
  - Virtualization of hardware resources (memory, CPU, peripherals ...)
  - System runs many isolated virtual machines

Full System Virtualization

- Idea: fake the entire system
  - Virtualization of hardware resources (memory, CPU, peripherals ...)
  - System runs many isolated virtual machines
- Implementation

- Idea: fake the entire system
  - Virtualization of hardware resources (memory, CPU, peripherals ...)
  - System runs many isolated virtual machines
- Implementation
  - Managed by hypervisor

Full System Virtualization

- Idea: fake the entire system
  - Virtualization of hardware resources (memory, CPU, peripherals ...)
  - System runs many isolated virtual machines
- Implementation
  - Managed by hypervisor
    - Xen, VMware, VirtualBox, Hyper-V, Qemu ...

Full System Virtualization

- Idea: fake the entire system
  - Virtualization of hardware resources (memory, CPU, peripherals ...)
  - System runs many isolated virtual machines
- Implementation
  - Managed by hypervisor
    - Xen, VMware, VirtualBox, Hyper-V, Qemu ...
  - Typically hardware-accelerated

Full System Virtualization

- Idea: fake the entire system
  - Virtualization of hardware resources (memory, CPU, peripherals ...)
  - System runs many isolated virtual machines
- Implementation
  - Managed by hypervisor
    - Xen, VMware, VirtualBox, Hyper-V, Qemu ...
  - Typically hardware-accelerated
  - CloudOS in the summer term

Full System Virtualization

- Idea: fake the entire system
  - Virtualization of hardware resources (memory, CPU, peripherals ...)
  - System runs many isolated virtual machines
- Implementation
  - Managed by hypervisor
    - Xen, VMware, VirtualBox, Hyper-V, Qemu ...
  - Typically hardware-accelerated
  - CloudOS in the summer term
- What if VM compromises hypervisor?

Full System Virtualization

- Idea: fake the entire system
  - Virtualization of hardware resources (memory, CPU, peripherals ...)
  - System runs many isolated virtual machines
- Implementation
  - Managed by hypervisor
    - Xen, VMware, VirtualBox, Hyper-V, Qemu ...
  - Typically hardware-accelerated
  - CloudOS in the summer term
- What if VM compromises hypervisor?
- Is there an end to this recursive problem?

Enclaves


ENCLAVES

- Observation: Sandboxes follow hierarchical ring model

- Observation: Sandboxes follow hierarchical ring model
  - Higher rings (kernel space) have strictly higher privileges

- Observation: Sandboxes follow hierarchical ring model
  - Higher rings (kernel space) have strictly higher privileges
  - Lower rings (user space) need to fully trust higher rings

- Observation: Sandboxes follow hierarchical ring model
  - Higher rings (kernel space) have strictly higher privileges
  - Lower rings (user space) need to fully trust higher rings
  - Vulnerability in higher ring is fatal

- Observation: Sandboxes follow hierarchical ring model
  - Higher rings (kernel space) have strictly higher privileges
  - Lower rings (user space) need to fully trust higher rings
  - Vulnerability in higher ring is fatal
- Idea: build a reverse sandbox: Enclaves

- Observation: Sandboxes follow hierarchical ring model
  - Higher rings (kernel space) have strictly higher privileges
  - Lower rings (user space) need to fully trust higher rings
  - Vulnerability in higher ring is fatal
- Idea: build a reverse sandbox: Enclaves
  - Only trust enclave code (and hardware)

- Observation: Sandboxes follow hierarchical ring model
  - Higher rings (kernel space) have strictly higher privileges
  - Lower rings (user space) need to fully trust higher rings
  - Vulnerability in higher ring is fatal
- Idea: build a reverse sandbox: Enclaves
  - Only trust enclave code (and hardware)
  - Distrust all non-enclave code

- Observation: Sandboxes follow hierarchical ring model
  - Higher rings (kernel space) have strictly higher privileges
  - Lower rings (user space) need to fully trust higher rings
  - Vulnerability in higher ring is fatal
- Idea: build a reverse sandbox: Enclaves
  - Only trust enclave code (and hardware)
  - Distrust all non-enclave code
    - Host application, kernel, hypervisor

- Observation: Sandboxes follow hierarchical ring model
  - Higher rings (kernel space) have strictly higher privileges
  - Lower rings (user space) need to fully trust higher rings
  - Vulnerability in higher ring is fatal
- Idea: build a reverse sandbox: Enclaves
  - Only trust enclave code (and hardware)
  - Distrust all non-enclave code
    - Host application, kernel, hypervisor
  - Example: Intel SGX/TDX, AMD-SEV, ARM Trustzone

- Properties

- Properties
  - Enclaves protect a piece of secure code / data

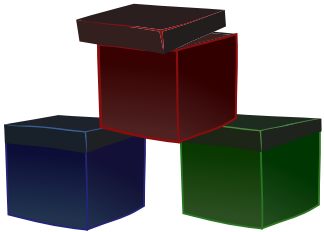
- Properties
  - Enclaves protect a piece of secure code / data
  - Enclaves cannot sandbox untrusted code

- Properties
  - Enclaves protect a piece of secure code / data
  - Enclaves cannot sandbox untrusted code
  - Can be (mis)used for Digital Rights Management (DRM), hiding malware

- Properties
  - Enclaves protect a piece of secure code / data
  - Enclaves cannot sandbox untrusted code
  - Can be (mis)used for Digital Rights Management (DRM), hiding malware
- Are we (too) secure now?

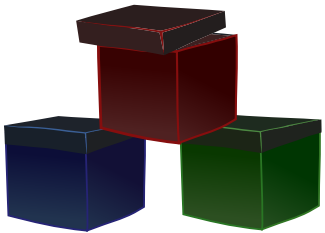
Summary & Outlook

Privilege Minimization Overview



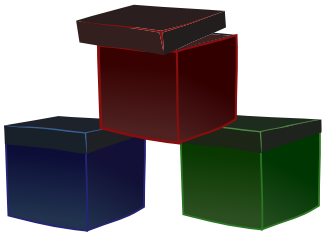
- Everything is a box

Privilege Minimization Overview



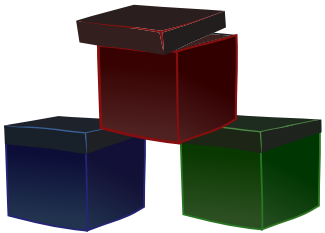
- Everything is a box
  - **Compartmentalization**: Make boxes as small as possible

Privilege Minimization Overview



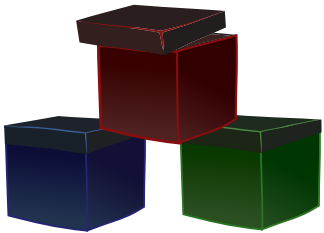
- Everything is a box
  - **Compartmentalization**: Make boxes as small as possible
  - **Isolation**: A box shall have minimal permission

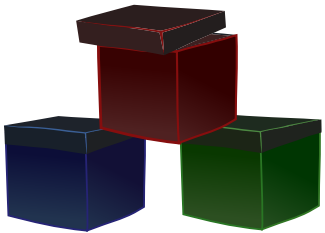
Privilege Minimization Overview



- Everything is a box
  - **Compartmentalization**: Make boxes as small as possible
  - **Isolation**: A box shall have minimal permission
  - "Principle of least privileges"

Isolation techniques

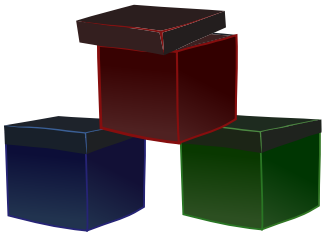




Isolation techniques

- In-process Sandboxing
  - Isolate domains within process

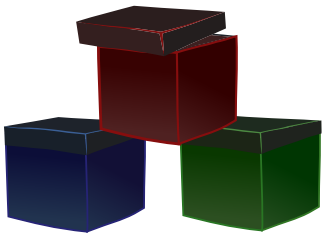
Privilege Minimization Techniques



Isolation techniques

- In-process Sandboxing
  - Isolate domains within process
- Process Sandboxing
  - Seccomp
  - Cgroups
  - Namespaces
  - Docker container = seccomp + cgroups + namespaces

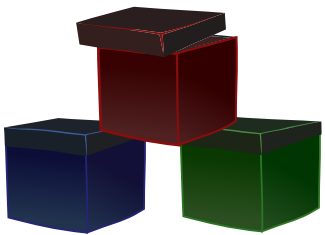
Privilege Minimization Techniques



Isolation techniques

- In-process Sandboxing
  - Isolate domains within process
- Process Sandboxing
  - Seccomp
  - Cgroups
  - Namespaces
  - Docker container = seccomp + cgroups + namespaces
- Virtualization
  - Full system virtualization

Privilege Minimization Techniques



Isolation techniques

- In-process Sandboxing
  - Isolate domains within process
- Process Sandboxing
  - Seccomp
  - Cgroups
  - Namespaces
  - Docker container = seccomp + cgroups + namespaces
- Virtualization
  - Full system virtualization
- Enclaves
  - Reverse sandbox

Summary

Recap: Defense In Depth

Attacker's perspective

- Vulnerability discovery ☑
- Exploitation ☑
- Privilege elevation ☑

Defender's perspective

- Vulnerability prevention ☑
- Exploit prevention ☑
- Privilege minimization ☑

Questions?


OUR DEVICES ARE NOW 100% SECURE.
I TURNED THEM ALL OFF.
HOW DID YOU DO THAT?
© D.Fletcher for CloudTweaks.com