Secure Software Development

Countermeasures: Privilege Minimization

Daniel Gruss, Vedad Hadzic, Andreas Kogler, Martin Schwarzl, Marcel Nageler

27.11.2020

Winter 2021/22, www.iaik.tugraz.at

Table of contents

1. Stack Buffer Overflows
2. Code Injection Attacks
3. Code Reuse Attacks
4. Summary

Attacker's perspective

- Vulnerability discovery ☑
- Exploitation ☑
- Privilege elevation ☑

Defender's perspective

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

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

- admin flag, spawn a shell, cat flag.txt, gain persistence

Defender's perspective

Vulnerability prevention

- Code quality, memory safety, type safety, error handling ...

Exploit prevention

- Compiler/runtime defenses, hardware defenses

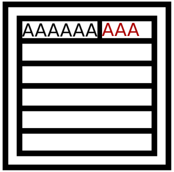
Privilege minimization

- System call filtering, sandboxing, virtualization

- 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!

- Arbitrary Code Execution
  - Kernel support
    - Syscall filtering
    - AppArmor
    - SeLinux
    - File system permissions
    -
  - Sandboxing
    - containers, jails
    - Software-based in-process: nacl/sfi: pointer masking in compiler, or binary rewriting
    - Emulation: pre-built quemu, dynamic code generation for performance, across architectures/ISAs
    - JavaScript Sandbox – abstract machines, dedicated callback hooks
    - Wasm Sandbox
    - HW-based in-process
    - Process-level sandbox: sandbox2
  - Virtualization

Stack Buffer Overflows



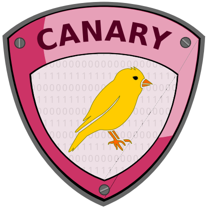

```
void printName(char* buffer) {
  char name[16];
  strcpy(name, buffer);
  printf("Hello %s\n", name);
}

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

- Observation 1: Most buffer overflows are 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

"This means something
but I can't remember what!"

- If the mine canary is dead, get out immediately



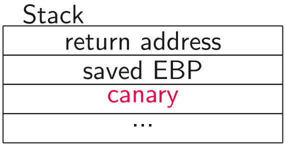
Idea: Introduce a canary on the stack that signals a hazard

- 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 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;
}
```

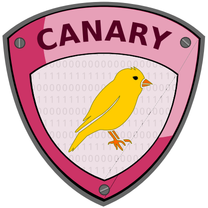
```
% gcc -o stack -fno-stack-protector stack.c
% ./stack AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[1]    12345 segmentation fault (core dumped)  ./stack
```

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

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

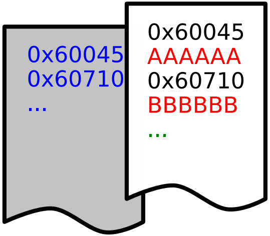


Properties

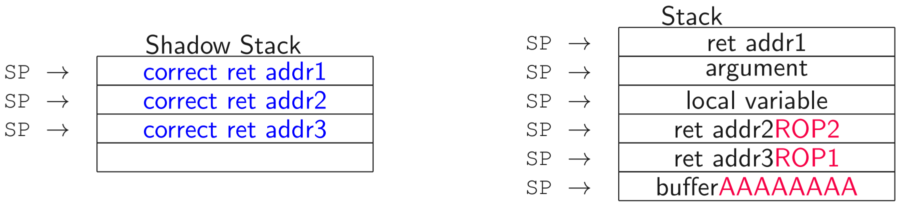
- Detects linear stack buffer overflows corrupting return address
- Does not detect
  - overflowing one buffer into the other
  - non-linear overflows, e.g. `buffer[8 * input];`
  - format string vulnerabilities ...
- Probabilistic defense
  - Ineffective if attacker can guess or leak the canary value
- 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

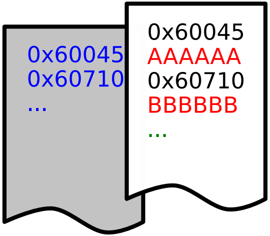


- Idea: duplicate return address on separate stack
- Implementation: compiler extension similar to canaries
  - Prologue: push return address also on shadow stack
  - Epilogue: verify return address before doing `retq`



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

Shadow Stack Summary

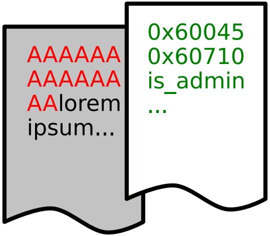


Properties

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

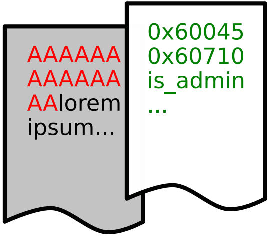
Why duplicate at all if we assume shadow stack is secure?

Safe Stack



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

Safe Stack Summary



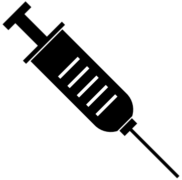
Properties

- 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?

- Control-flow Enforcement Technology (CET) for Intel (and AMD)

Code Injection Attacks



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

Data Execution Prevention

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

- 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
(gdb) run
Starting program: /home/shellcode
len:27 bytes

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

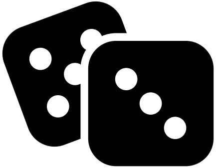
```
% execstack -s ./shellcode
% gdb ./shellcode
(gdb) run
Starting program: /home/shellcode
len:27 bytes
process 9494 is executing new program: /bin/dash
$
```

```
% readelf -e ./shellcode
Program Headers:
  Type        Offset     VirtAddr   PhysAddr
              FileSiz    MemSiz      Flags  Align
  LOAD        0x00000000 0x00400000 0x0000400000 \ code is
              0x0000075c 0x0000075c  R E   200000 / read-execute
  LOAD        0x00000e10 0x00600e10 0x0000600e10 \ data is
              0x0000024c 0x00000250  RW    200000 / read-write
  ...
  GNU_STACK 0x00000000 0x00000000 0x0000000000 \ we made stack
              0x00000000 0x00000000  RWE   10      / read-write-exec
```

Data Execution Prevention Summary

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

Code Reuse Attacks



- 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

- 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`

```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
```

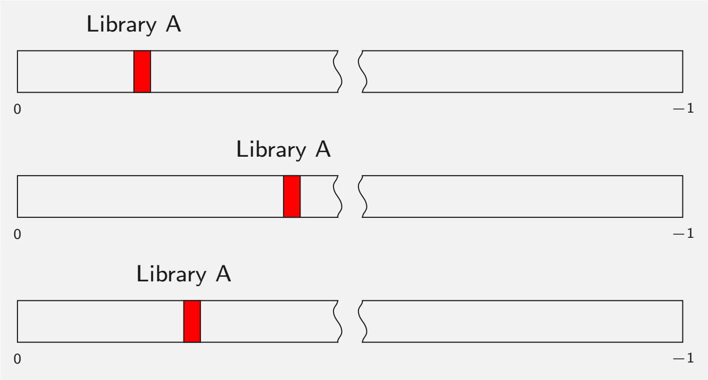
```
% cat /proc/self/maps
00400000-0040c000 r-xp 00000000 fd:00 395191           /bin/cat
0060b000-0060c000 r--p 0000b000 fd:00 395191           /bin/cat
0060c000-0060d000 rw-p 0000c000 fd:00 395191           /bin/cat
00b0c000-00b2d000 rw-p 00000000 00:00 0                [heap]
7efcbb558000-7efcbb87e000 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]
```

```
% 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 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
0129e000-012bf000 rw-p 00000000 00:00 0                [heap]
7f0b4f569000-7f0b4f88f000 r--p 00000000 fd:00 11534857 /usr/lib/locale/locale-archive
7f0b4f88f000-7f0b4fa4f000 r-xp 00000000 fd:00 4587769  /lib/x86_64-linux-gnu/libc-2.23.so
7f0b4fa4f000-7f0b4fc4f000 ---p 001c0000 fd:00 4587769  /lib/x86_64-linux-gnu/libc-2.23.so
7f0b4fc4f000-7f0b4fc53000 r--p 001c0000 fd:00 4587769  /lib/x86_64-linux-gnu/libc-2.23.so
7f0b4fc53000-7f0b4fc55000 rw-p 001c4000 fd:00 4587769  /lib/x86_64-linux-gnu/libc-2.23.so
7f0b4fc55000-7f0b4fc59000 rw-p 00000000 00:00 0
7f0b4fc59000-7f0b4fc7f000 r-xp 00000000 fd:00 4588089  /lib/x86_64-linux-gnu/ld-2.23.so
7f0b4fe49000-7f0b4fe4c000 rw-p 00000000 00:00 0
7f0b4fe5c000-7f0b4fe7e000 rw-p 00000000 00:00 0
7f0b4fe7e000-7f0b4fe7f000 r--p 00025000 fd:00 4588089  /lib/x86_64-linux-gnu/ld-2.23.so
7f0b4fe7f000-7f0b4fe80000 rw-p 00026000 fd:00 4588089  /lib/x86_64-linux-gnu/ld-2.23.so
7f0b4fe80000-7f0b4fe81000 rw-p 00000000 00:00 0
7fff50016000-7fff50037000 rw-p 00000000 00:00 0        [stack]
7fff500d9000-7fff500db000 r--p 00000000 00:00 0        [vvar]
7fff500db000-7fff500dd000 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
- ❓ How does randomized code remain functional?

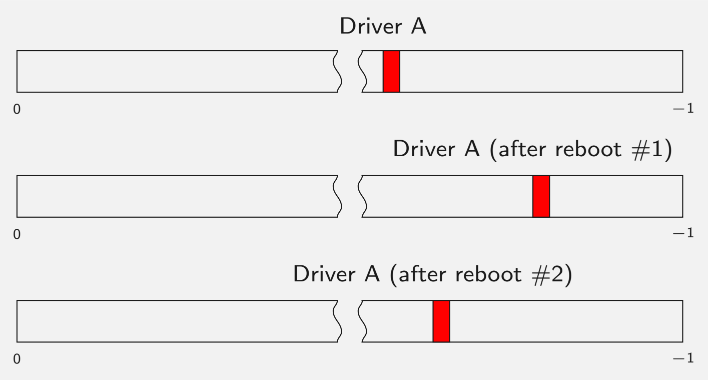
- Within a module
  - Compiler replaces absolute addresses with (rip-)relative addresses
  - Code can be executed from virtually any offset
  - Shared libraries: compile flags `-fpic`, `-fPIC`
  - Executable: compile flags `-fpie`, `-fPIE`
- Across modules
  - Runtime linker resolves addresses via:
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

```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
```

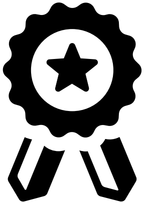
```c
#include <stdio.h>

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

```
% gcc -o main -fPIE main.c
% objdump -d main
0000000000400526 <main>:
  400526: 55                    push   %rbp
  400527: 48 89 e5              mov    %rsp,%rbp
  40052a: 48 8d 3d 93 00 00 00  lea    0x93(%rip),%rdi    # Address of Hi
  400531: e8 ca fe ff ff        callq  400400 <puts@plt> # PLT
  400536: b8 00 00 00 00        mov    $0x0,%eax
  40053b: 5d                    pop    %rbp
  40053c: c3                    retq
```



- Same driver module is randomized to different offset at each boot
- Leaking kernel or driver addresses defeats KASLR

- Properties
  - Super cheap
  - Make exploit development hard
  - 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)



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 Ubuntu 20.04.1 LTS, kernel 4.19.0-11

Diversify

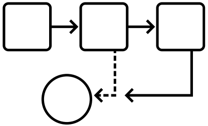
Diversification

- Idea: randomize compiler output
  - Each user/server has a different binary
  - Attacker needs to redevelop exploit for each binary
- Implementation
  - Light-weight randomization via reordering of functions, memory objects, basic blocks ...
  - Heavy-weight obfuscation techniques

Properties

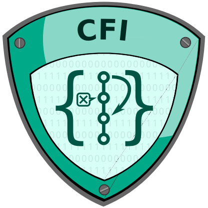
- Break portability of exploits, thus dampering large-scale exploitation
- Does not prevent exploitation!
- Counteracts reproducible builds
- Counteracts debugging
- No wide adoption yet

CFI



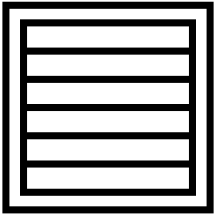
- Observation: most attacks corrupt code pointers
- Idea: specifically protect code pointers
    - CFI: Program must stay inside control flow graph (CFG)
    - Attacker cannot (arbitrarily) change control flow
    - Prevent ROP and maybe ret2libc
- Code pointers everywhere
    - Return addresses
   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

Control Flow Integrity (CFI)

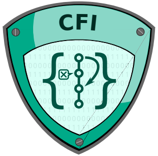


- CFI on backward edges
  - Return addresses shall only point to the real caller
  - Solved via shadow stack / safe stack
- CFI on forward edges
  - GOT entries shall only contain legitimate pointers
  - Function pointers shall only call legitimate functions
  - C++ vtable shall only call legitimate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

Global Offset Table (GOT) Protection

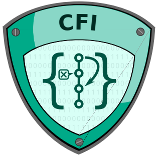


- Observation: only runtime linker populates GOT
  - Any other GOT manipulation is either an accident or an attack
- Idea: Make GOT read-only
- Implementation
  - Linker populates *all* GOT entries at program start → slowdown
  - Compiler flag `-Wl,-z,relro` "relocations read-only"



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

- 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

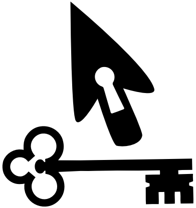
- Every function entry marked with `endbr64` instruction
- Call instructions only succeed towards `endbr64` instructions
- Supported by future Intel and AMD CPUs

```
% 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
    ...
```

```
#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;
}
```

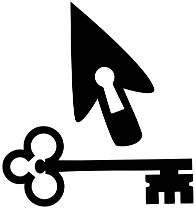
```
% ./test
A
B
secret
```

```
% 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
              ^
```



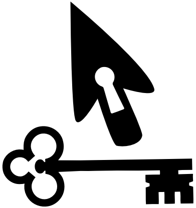
ARMv8.3 hardware-based pointer authentication

- 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 message authentication code (MAC)
    - Novel crypto algorithm "QARMA"
  - Hardware stores parts of the MAC in unused pointer bits
  - Before dereferencing (calling) a pointer, compiler authenticates it using special `AUT` instruction
  - Hardware invalidates pointer on authentication failure



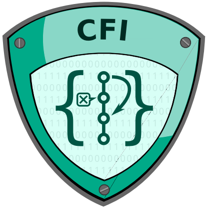
MAC algorithm has three inputs

- Pointer value
- Secret (process-dependent) key
- Additional *context* controllable via the instructions
- *Context* can be used to distinguish
  - Function signatures
  - Types of data pointers
  - Stack frames
  - ...



☆ Properties

- Cryptographically-enforced CFI
- Effectiveness depends on additional *context* input
  - Precision: if *context* is zero, attacker can exchange all authenticated pointers (Similar security as 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



☆ Properties

- CFI: attacker cannot escape control flow graph (CFG)
  - Defeats ROP
  - 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?

Summary

Arbitrary Code Execution

Kernel support

- Syscall filtering
- AppArmor
- SeLinux
- File system permissions
-

Sandboxing

- containers, jails
- Software-based in-process: nacl/sfi: pointer masking in compiler, or binary rewriting
- Emulation: pre-built quemu, dynamic code generation for performance, across architectures/ISAs
- JavaScript Sandbox – abstract machines, dedicated callback hooks
- Wasm Sandbox
- HW-based in-process
- Process-level sandbox: sandbox2

Virtualization

Recap: Defense In Depth

Attacker's perspective

- Vulnerability discovery ☑
- Exploitation ☑
- Privilege elevation ☑

Defender's perspective

- Vulnerability prevention ☑
- Exploit prevention ☑
- Privilege minimization (next time)

Questions?

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