CS 453/698: Software and Systems Security

Module: An In-depth Study of Memory Errors
Lecture: Exploit mitigation

Meng Xu *(University of Waterloo)*

Winter 2025

Outline

Software security landscape

Generally speaking, almost all work in the software security area can be categorized into four bins:

- Vulnerability: *Identify a bug in the program that may cause some damage*
  - $f(Code) \rightarrow Bug$
- Exploitation: *Given a set of bugs, exploit them to achieve a desired goal*
  - $f(Code, \{...Bug...\}, Goal) \rightarrow Action$
- **Mitigation**: *Given a set of bugs and an associated set of exploits, prevent them*
  - $f(Code, \{...Bug...\}, \{...Action...\}) \rightarrow Blockage$
- Detection: *Given a program, check the existence of a specific type of bug*
  - $f(Code, Bug, [Action]) \rightarrow Signal$
- Prevention: *It is impossible to create a program that has a specific type of bug*

Themes of mitigation

- Principle of least privileges (PoLP)
- Reference monitoring / program shepherding
- Moving-target defense

Themes of mitigation

- Principle of least privileges (PoLP)
  - reduce permissions unless absolutely needed
- Reference monitoring / program shepherding
- Moving-target defense

Themes of mitigation

- Principle of least privileges (PoLP)
  - reduce permissions unless absolutely needed
- Reference monitoring / program shepherding
  - keep an eye on the program while it is executing
- Moving-target defense

Themes of mitigation

- Principle of least privileges (PoLP)
  - reduce permissions unless absolutely needed
- Reference monitoring / program shepherding
  - keep an eye on the program while it is executing
- Moving-target defense
  - non-determinism is useful in software security when
    * it has no impact on the intended finite state machine BUT
    * limits attackers' abilities to program the weird machine.

Outline

DEP a.k.a., W$\oplus$X

DEP a.k.a., W⊕X

DEP – Data Execution Prevention

W⊕X – Write exclusive-or eXecute

You can either write data **OR** execute code in a memory region, but **never both**.

DEP a.k.a., W⊕X

DEP – Data Execution Prevention

W⊕X – Write exclusive-or eXecute

You can either write data **OR** execute code in a memory region, but **never both**.

Implementation: `gcc -z execstack`.

Motivation for type-based heap allocation

Motivation for type-based heap allocation

A more realistic use-after-free (UAF) exploit:

```c
struct N {
  long user;
  int (*fn)(void);
};
```

```c
struct O {
  int (*oper)(void);
  long id;
};
```

```c
void foo(long user) {
  struct N *p =
    malloc(sizeof(struct N));

  p->fn = __safe_function_1;
  p->user = user;

  /* ... */
  /* later in the code */
  /* ... */
  p->fn();
}
```

```c
void bar(long id) {
  struct O *x =
    malloc(sizeof(struct O));

  x->oper = __safe_function_2;
  x->id = id;
  struct O *q = x;
  free(x);            // q is dangling

  /* later in the code */
  q->oper();
}
```

Sample UAF-exploit (continued)

```
{
  /* from bar(..) */
  struct O *x =
    malloc(sizeof(struct O));

  x->oper = __safe_function_2;
  x->id = id;
  struct O *q = x;
  free(x);              // q is dangling

  /* from foo(..) */
  struct N *p =
    malloc(sizeof(struct N));

  p->fn = __safe_function_1;
  p->user = user;

  /* from bar(..) */
  q->oper();
}
```

Type-based heap allocation

If a memory address refers to a heap object of type `T`, it will **always** refer to objects of type `T`, no matter what (e.g., freed and re-allocated).

NOTE: this does not imply that this memory address will be assigned to a `T *` pointer. It can be assigned to a `void *`, an `int *`, or anything.

Outline

CFI: introduction

Control-Flow Integrity (CFI) is a classic example of runtime reference monitor in software security.

CFI: introduction

Control-Flow Integrity (CFI) is a classic example of runtime reference monitor in software security.

CFI is also sometimes referred to as program shepherding

> *monitoring control flow transfers during program execution to enforce a security policy — from a paper in USENIX Security'02.*

Basic use cases of CFI

```
void f1();
void f2();
void f3();
void f4(int, int);

void foo(int usr) {
  void (*func)();

  if (usr == MAGIC)
    func = f1;
  else
    func = f2;

  // forward edge CFI check
  CHECK_CFI_FORWARD(func);
  func();

  // backward edge CFI check
  CHECK_CFI_BACKWARD();
}
```

Basic use cases of CFI

```
void f1();
void f2();
void f3();
void f4(int, int);

void foo(int usr) {
  void (*func)();

  if (usr == MAGIC)
    func = f1;
  else
    func = f2;

  // forward edge CFI check
  CHECK_CFI_FORWARD(func);
  func();

  // backward edge CFI check
  CHECK_CFI_BACKWARD();
}
```

Option 1: allow all functions

- f1, f2, f3, f4, foo, printf, system, ...

Basic use cases of CFI

```
void f1();
void f2();
void f3();
void f4(int, int);

void foo(int usr) {
  void (*func)();

  if (usr == MAGIC)
    func = f1;
  else
    func = f2;

  // forward edge CFI check
  CHECK_CFI_FORWARD(func);
  func();

  // backward edge CFI check
  CHECK_CFI_BACKWARD();
}
```

Option 1: allow all functions

- f1, f2, f3, f4, foo, printf, system, ...

Option 2: allowed only functions defined in the current module

- f1, f2, f3, f4, foo

Basic use cases of CFI

```c
void f1();
void f2();
void f3();
void f4(int, int);

void foo(int usr) {
  void (*func)();

  if (usr == MAGIC)
    func = f1;
  else
    func = f2;

  // forward edge CFI check
  CHECK_CFI_FORWARD(func);
  func();

  // backward edge CFI check
  CHECK_CFI_BACKWARD();
}
```

Option 1: allow all functions

- f1, f2, f3, f4, foo, printf, system, ...

Option 2: allowed only functions defined in the current module

- f1, f2, f3, f4, foo

Option 3: allow functions with type signature `void (*)()`

- f1, f2, f3

Basic use cases of CFI

```c
void f1();
void f2();
void f3();
void f4(int, int);

void foo(int usr) {
  void (*func)();

  if (usr == MAGIC)
    func = f1;
  else
    func = f2;

  // forward edge CFI check
  CHECK_CFI_FORWARD(func);
  func();

  // backward edge CFI check
  CHECK_CFI_BACKWARD();
}
```

Option 1: allow all functions

- f1, f2, f3, f4, foo, printf, system, ...

Option 2: allowed only functions defined in the current module

- f1, f2, f3, f4, foo

Option 3: allow functions with type signature `void (*)()`

- f1, f2, f3

Option 4: allow functions whose address are taken (e.g., assigned)

- f1, f2

Example: Microsoft Return-flow Guard (RFG)

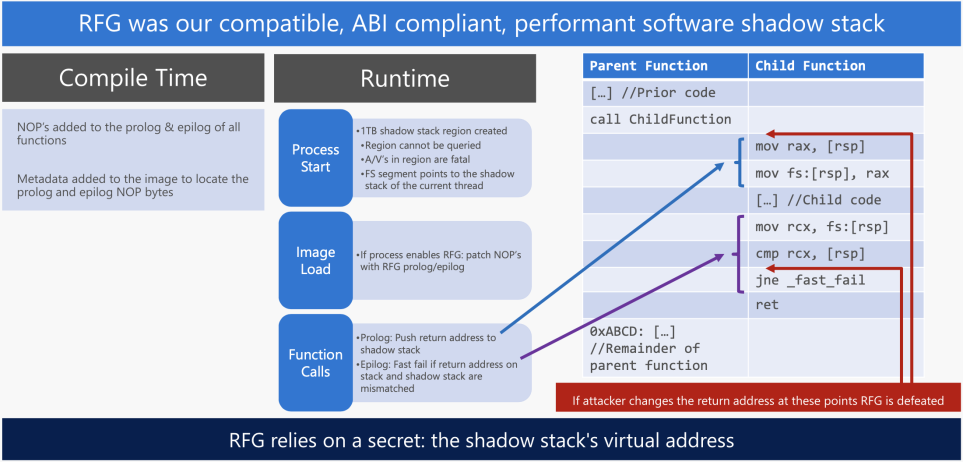
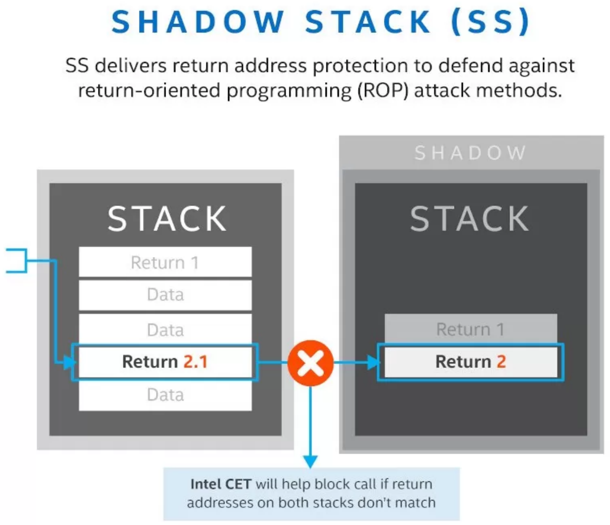


Illustration taken from Microsoft Talk: The Evolution of CFI Attacks and Defenses

Back-edge protection: shadow stack



Copyright: Intel

CET: shadow stack

- For every regular stack CET adds a shadow stack region, which is indexed via a new register `%ssp`.
- Regular memory stores (executed from any ring) are not allowed in shadow stack region

When enabled,

- Each time a `call` instruction gets executed, in addition to the return address being pushed onto the regular stack, a copy of it is also pushed (automatically) onto the shadow stack.
- Each time a `ret` instruction gets executed, the return addresses pointed by `%rsp` and `%ssp` are (automatically) popped from the two stacks, and their values are compared together.

CET: Indirect Branch Tracking (IBT)

CET introduces a new (4-byte) instruction, i.e., `endbr`, which becomes the **only** allowed target of indirect `call`/`jmp` instructions.

In other words, forward-edge transfers via (indirect) `call` or `jmp` instructions are pinned to code locations that are "marked" with an `endbr`; else, an exception (`#CP`) is raised.

IBT example

```
void main() {
    int (*f) {};
    f = foo;
    f();
}

int foo() {
    return 0;
}
```

```
<main>:
movq    $0x4004fb, -8(%rbp)
mov     -8(%rbp), %rdx
call    *%rdx
:
retq

<foo>:
endbr64
:
mov     rax, 0
:
retq
```

IBT example

```c
void main() {
    int (*f) {};
    int (*g) {};
    f = foo;
    g = bar;
    f();
    g();
}

int foo() {
    return 0;
}

int bar() {
    return 1;
}
```

```
<main>:
movq     $0x4004fb, -16(%rbp)
mov      -16(%rbp), %rdx
call     *%rdx
mov      -8(%rbp), %rdx
call     *%rdx
:
retq

<foo>:
endbr64
:
mov      rax, 0
:
retq

<bar>:
endbr64
:
mov      rax, 1
:
retq
```

Security boundaries of CFI-protected programs

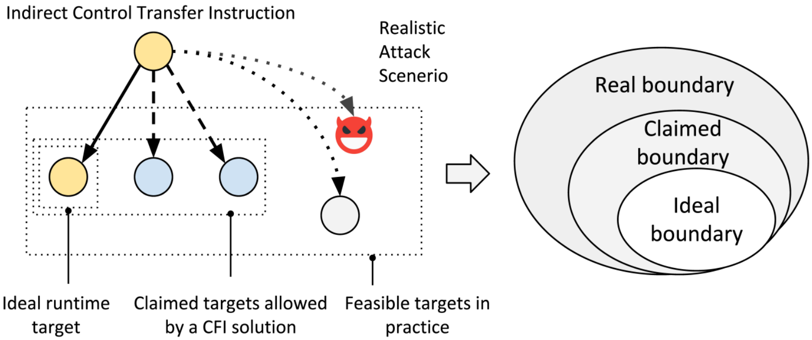


Figure from a paper published in ACM CCS'20

Pointer integrity

Goal: ensures pointers in memory remain unchanged.

Pointer integrity

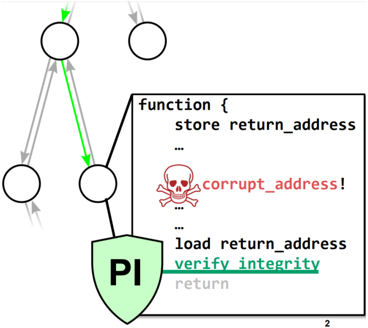
Goal: ensures pointers in memory remain unchanged.

- i.e., the value of the pointer remains unchanged, not the memory content referred to by this pointer.

Pointer integrity

Goal: ensures pointers in memory remain unchanged.

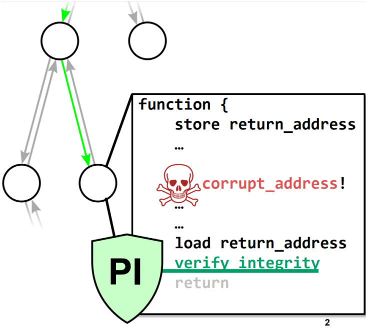
- i.e., the value of the pointer remains unchanged, not the memory content referred to by this pointer.
- Perfect code pointer integrity implies control-flow integrity (CFI).



Pointer integrity

Goal: ensures pointers in memory remain unchanged.

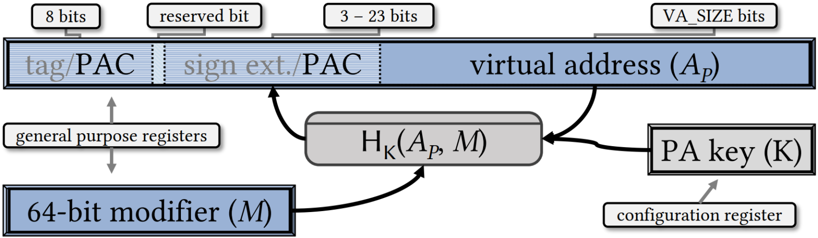
- i.e., the value of the pointer remains unchanged, not the memory content referred to by this pointer.
- Perfect code pointer integrity implies control-flow integrity (CFI).



- Data pointer integrity is also important (e.g., against data-only attacks and data-oriented programming) and can be (partially) achieved via Pointer Authentication.

Overview of Arm Pointer Authentication (PA)

Available since Armv8.3-A instruction set architecture (ISA) when the processor executes in 64-bit Arm state (AArch64)



PA consists of a set of instructions for creating and authenticating pointer authentication codes (PACs).

PAC details

- Each PAC is derived from
  - A pointer value
  - A 64-bit context value (modifier)
  - A 128-bit secret key

PAC details

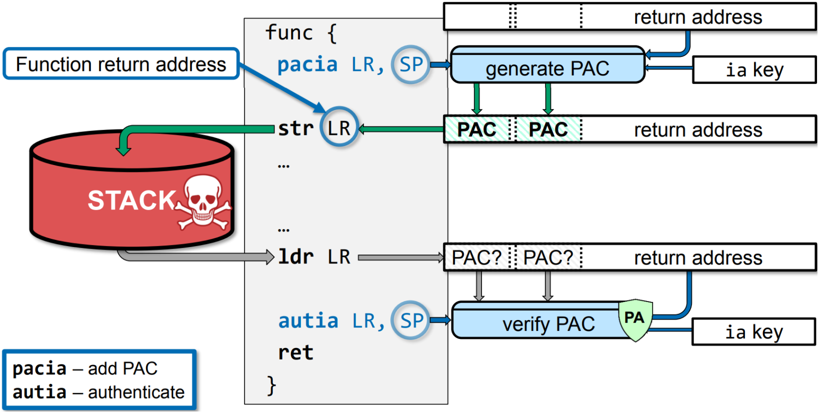
- Each PAC is derived from
  - A pointer value
    * an N-bit memory address
  - A 64-bit context value (modifier)
    * doesn't need to secret, as long as it provides enough entropy
  - A 128-bit secret key
    * held in system registers, set by the kernel per each process,
    * can be used, but cannot be read/written by userspace

PAC details

- Each PAC is derived from
  - A pointer value
    * an N-bit memory address
  - A 64-bit context value (modifier)
    * doesn't need to secret, as long as it provides enough entropy
  - A 128-bit secret key
    * held in system registers, set by the kernel per each process,
    * can be used, but cannot be read/written by userspace
- PAC essentially a key-ed message authentication code (MAC) where the MAC algorithm can be implementation defined
  - by default, it is QARMA
- Instructions hide the algorithm details (sign + authenticate)

Example: PA-based return address signing

Deployed as `-msign-return-address` in GCC and LLVM/Clang



$\langle$ **End** $\rangle$