CS 453/698: Software and Systems Security

Module: An In-depth Study of Memory Errors

Lecture: Exploit mitigation

Meng Xu *(University of Waterloo)*

Spring 2025

Outline

1. Introduction: what is mitigation?
2. Principle of least privileges (PoLP)
3. Reference monitoring
4. Moving-target defense

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$\oplus$X

DEP – Data Execution Prevention

W$\oplus$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

```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

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

```
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)

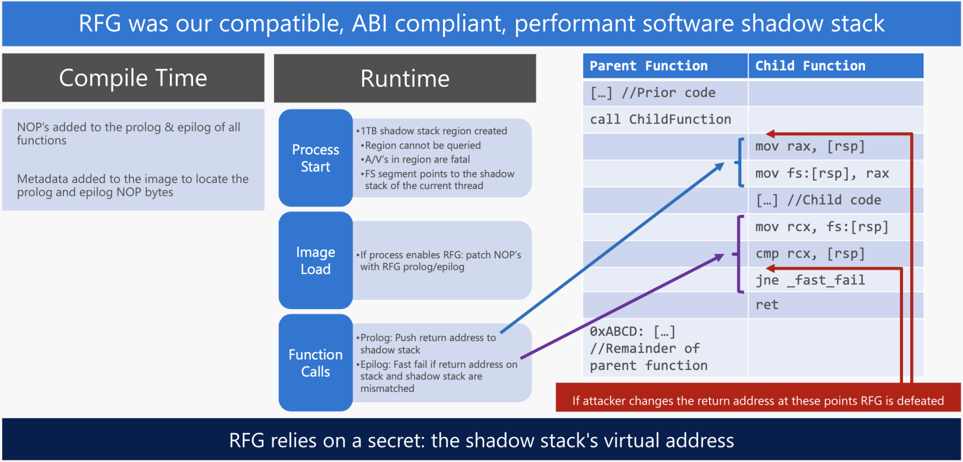
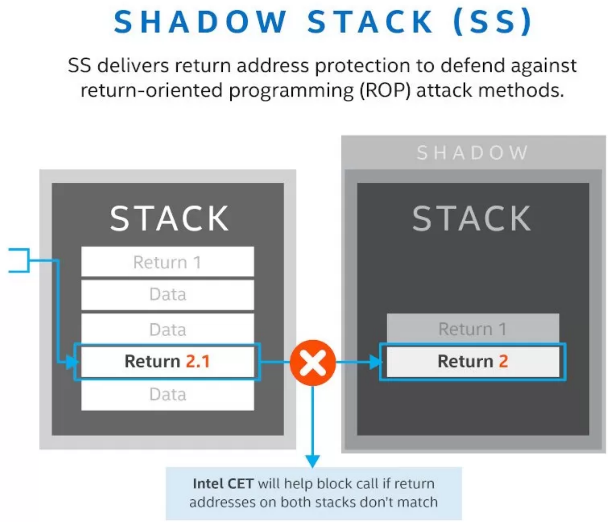


Illustration taken from Microsoft Talk: The Evolution of CFI Attacks and Defenses

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

```c
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

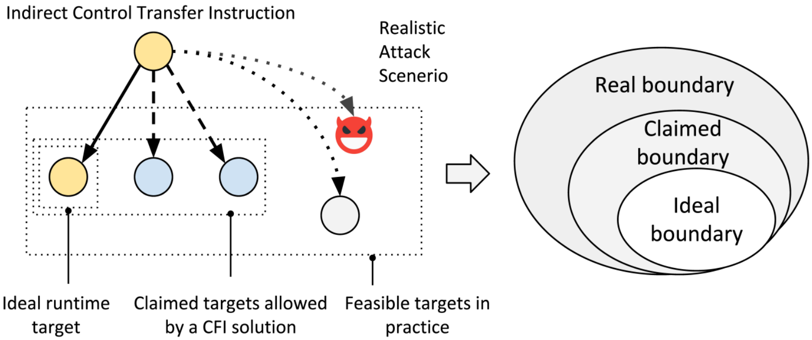


Figure from a paper published in ACM CCS'20

Pointer integrity

Goal: ensures pointers in memory remain unchanged.

Pointer integrity

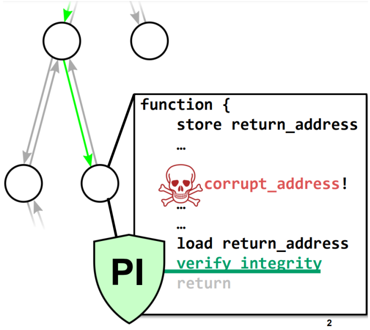
Goal: ensures pointers in memory remain unchanged.

- 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.

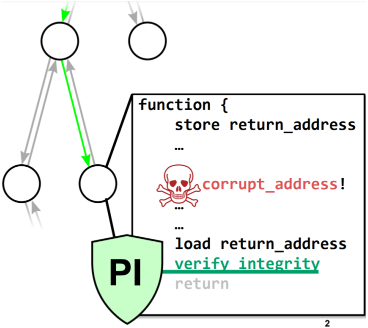
- i.e., the value of the pointer remains unchanged, not the memory content referred to by this pointer.
- Perfect code pointer integrity implies control-flow integrity (CFI).



Pointer integrity

Goal: ensures pointers in memory remain unchanged.

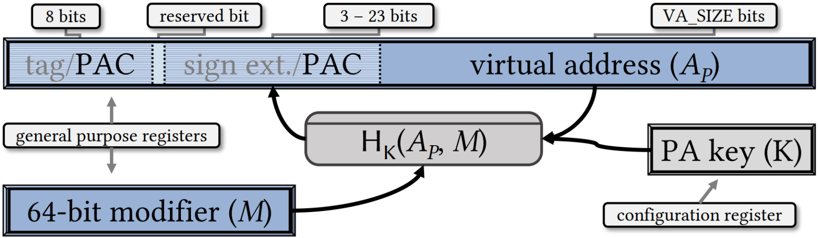
- i.e., the value of the pointer remains unchanged, not the memory content referred to by this pointer.
- Perfect code pointer integrity implies control-flow integrity (CFI).



- 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

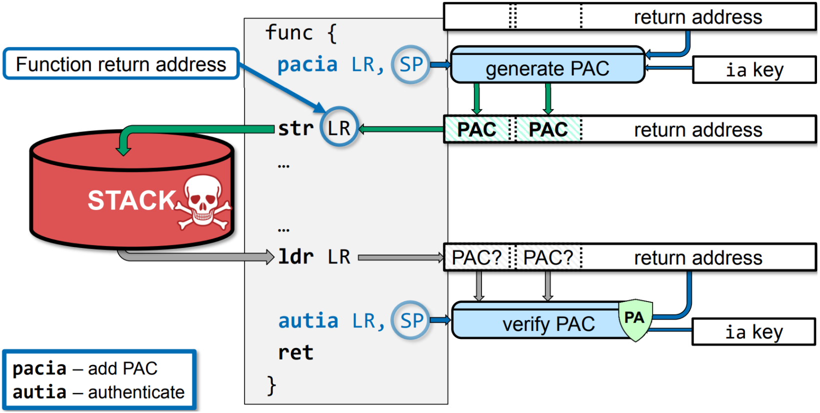
- Each PAC is derived from
  - A pointer value
    - an N-bit memory address
  - A 64-bit context value (modifier)
    - doesn't need to secret, as long as it provides enough entropy
  - A 128-bit secret key
    - held in system registers, set by the kernel per each process,
    - can be used, but cannot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



Outline

Why entropy in security?

Nondeterminism 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.

Why entropy in security?

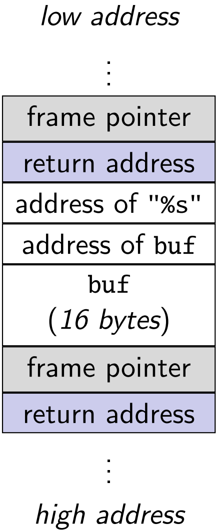
Nondeterminism is useful in software security when

- it has no impact on the intended finite state machine BUT
- limits attackers' abilities to program the weird machine.

In the rest of this lecture: we will examine some standard / deployed practices of safely introducing nondeterminism to boost system and software security.

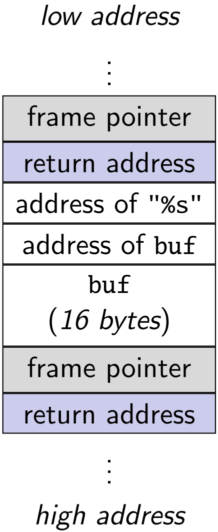
Recap: stack overflow

```c
int main() {
  char buf[16];
  scanf("%s", buf);
}
```



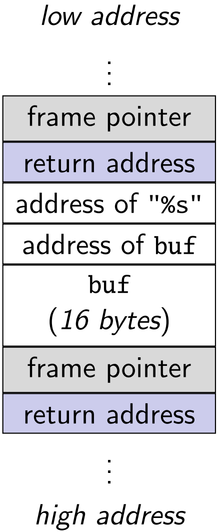
Stack canary intuition

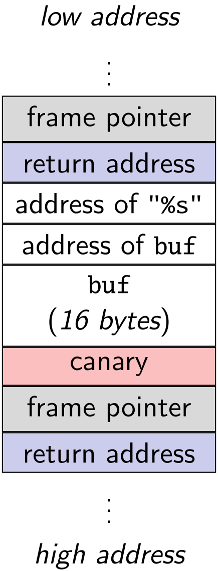
```c
int main() {
  char buf[16];
  scanf("%s", buf);
}
```



Stack canary intuition

```c
int main() {
  char buf[16];
  scanf("%s", buf);
}
```

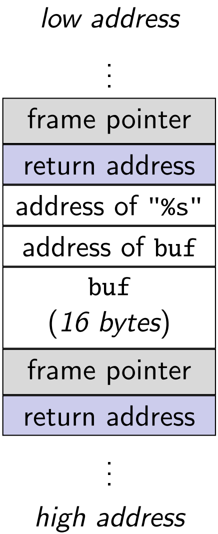


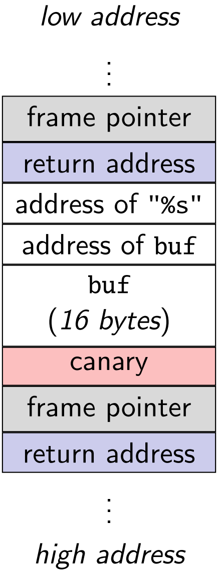


Stack canary intuition

```c
int main() {
  char buf[16];
  scanf("%s", buf);
}
```

- On function entry, push canary value X onto stack.
- On function return, check canary value is still X.





Original use of canary

Figure: Canaries in coal-mining. Credits / Trademark: Alamy Stock Photo

The default implementation in GCC

```c
int main() {
  char buf[16];
  scanf("%s", buf);
}
```

```c
extern uintptr_t __stack_chk_guard;
noreturn void __stack_chk_fail(void);

int main() {
  uintptr_t canary = __stack_chk_guard;

  char buf[16];
  scanf("%s", buf);

  if ((canary = canary ^ __stack_chk_guard) != 0) {
    __stack_chk_fail();
  }
}
```

The default implementation in GCC

```c
int main() {
  char buf[16];
  scanf("%s", buf);
}
```

```c
extern uintptr_t __stack_chk_guard;
noreturn void __stack_chk_fail(void);

int main() {
  uintptr_t canary = __stack_chk_guard;

  char buf[16];
  scanf("%s", buf);

  if ((canary = canary ^ __stack_chk_guard) != 0) {
    __stack_chk_fail();
  }
}
```

- The `__stack_chk_guard` and `__stack_chk_fail` symbols are normally supplied by a GCC library called `libssp`.
- You also have the option of specifying your own value for stack canaries.

Design choices of stack canaries

Design choices of stack canaries

- Which value should we use as canary?
  - deterministic? secret? random?

Design choices of stack canaries

- Which value should we use as canary?
  - deterministic? secret? random?
- What is the granularity of the canary invocation?
  - per function? per execution?

Design choices of stack canaries

- Which value should we use as canary?
  - deterministic? secret? random?
- What is the granularity of the canary invocation?
  - per function? per execution?
- When to do the integrity check?
  - on function return? is that enough?

Design choices of stack canaries

- Which value should we use as canary?
  - deterministic? secret? random?
- What is the granularity of the canary invocation?
  - per function? per execution?
- When to do the integrity check?
  - on function return? is that enough?
- How much randomness is needed?
  - 1 byte? 8 bytes? 64 bytes?

Limitations of stack canary

- Vulnerable to information leak
  - e.g., using a buffer over read to retrieve the canary value

Limitations of stack canary

- Vulnerable to information leak
  - e.g., using a buffer over read to retrieve the canary value
- Limited protection for frame pointer and return address only
  - other stack variables are not protected

Limitations of stack canary

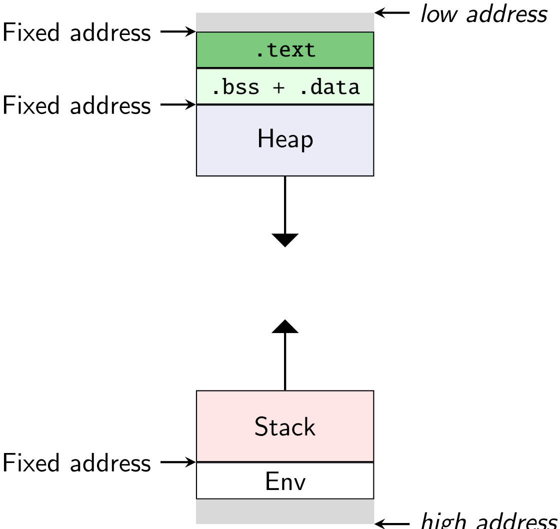
- Vulnerable to information leak
  - e.g., using a buffer over read to retrieve the canary value
- Limited protection for frame pointer and return address only
  - other stack variables are not protected
- Unable to defend against arbitrary writes
  - i.e., non-continuous overrides

Randomize the addresses

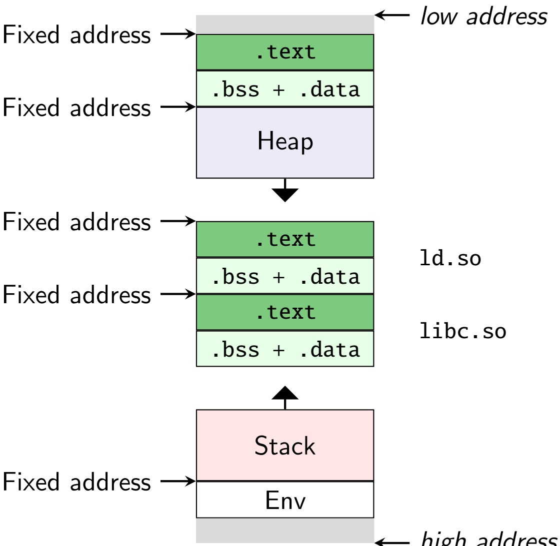
ASLR — Address Space Layout Randomization, is a system-level protection that randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries.

PIE — Position Independent Executable, is a body of machine code that executes properly regardless of its absolute address. This is also known as position-independent code (PIC).

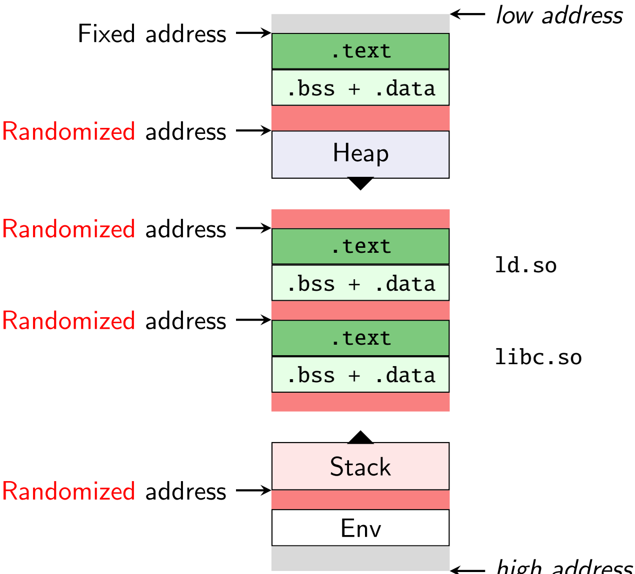
Base case: static program



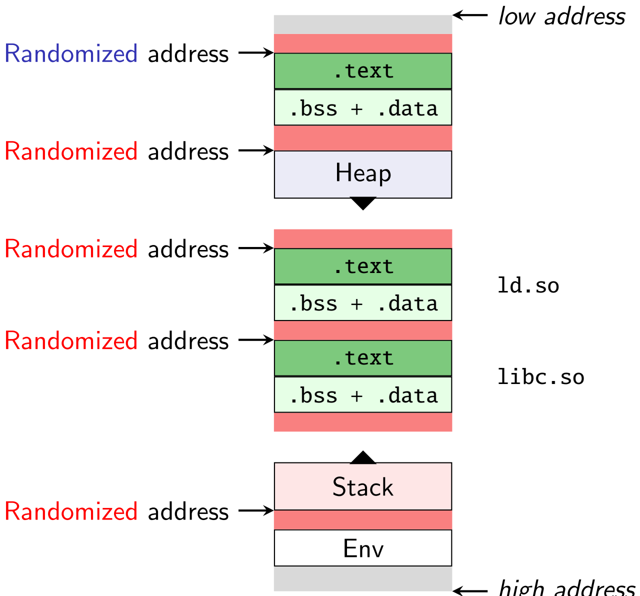
Static program + shared libraries



Static program + shared libraries + ASLR



Static program + shared libraries + ASLR + PIE



Paranoid randomization

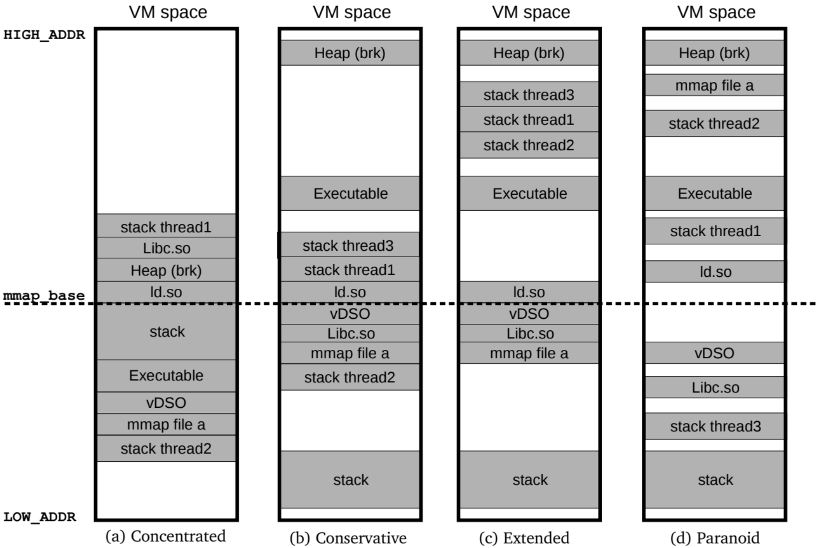


Figure: Different level of randomization proposed by the ASLR-NG project

Limitations of ASLR + PIE

- Limited entropy
  - visualized by the ASLR-NG project

Limitations of ASLR + PIE

- Limited entropy
  - visualized by the ASLR-NG project
- Memory layout inheritance
  - Child processes inherit/share the memory layout of the parent.

Motivation for secure heap allocators

Memory errors are equally (if not more) likely to happen on heap objects (compared with stack objects) which can cause all sorts of unexpected behaviors.

A heap buffer overflow case

```c
struct dispatcher {
    uint64_t counter;
    int (*action)(uint64_t counter, char *data);
}

int main() {
  char *p1 = malloc(16);
  char *p2 = malloc(sizeof(struct dispatcher));
  p2->counter = 0;
  p2->action = /* some valid function */;

  scanf("%s", p1);
  int result = p2->action(p2->counter, p1);

  free(p1);
  free(p2);
  return result;
}
```

A heap use-after-free case

```c
struct dispatcher {
  uint64_t counter;
  int (*action)(uint64_t counter, char *data);
}

char *p1;

void main() {
  p1 = malloc(16);
  pthread_create(/* ... */, thread_1);
  pthread_create(/* ... */, thread_2);
  /* wait for thread termination */
}
```

```c
void thread_1() {
  scanf("%15s", p1);
  /* ... compromised here ... */
  /* use-after-free */
  free(p1);
  ((struct dispatcher *)p1)
    ->action = /* bad function */;
}
```

```c
void thread_2() {
  char *p2 = malloc(
    sizeof(struct dispatcher));
  p2->counter = 0;
  p2->action = /* good function */;
  p2->action(p2->counter, p1);
  free(p2);
}
```

Secure heap allocators

These exploits have implicit assumptions on the layout of the heap, which can be invalidated by a secure heap allocator.

Basic allocator example

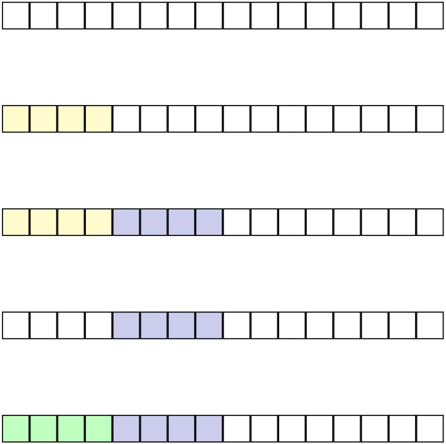
Initial state:

`p1 = malloc(16);`

`p2 = malloc(sizeof(..));`

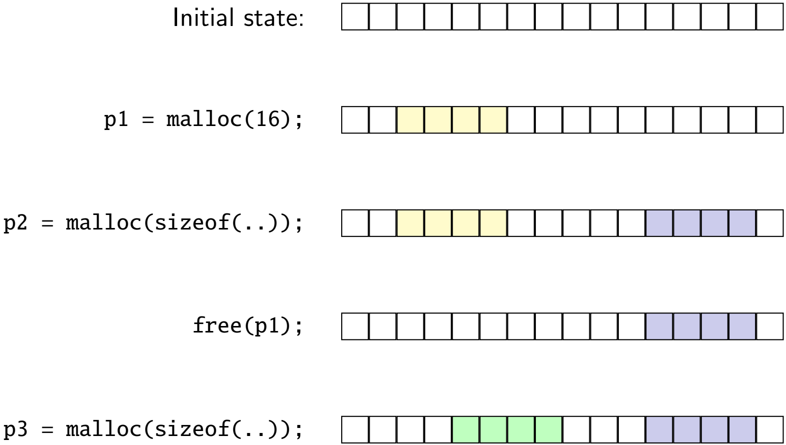
`free(p1);`

`p3 = malloc(sizeof(..));`



[0] Each square is a 4-byte box

Allocator + random placement



[0] Each square is a 4-byte box

Allocator + random placement + canary

Initial state:

```
p1 = malloc(16);

p2 = malloc(sizeof(..));

free(p1);

p3 = malloc(sizeof(..));
```

[0] Each square is a 4-byte box

Intuition: gene/DNA diversity

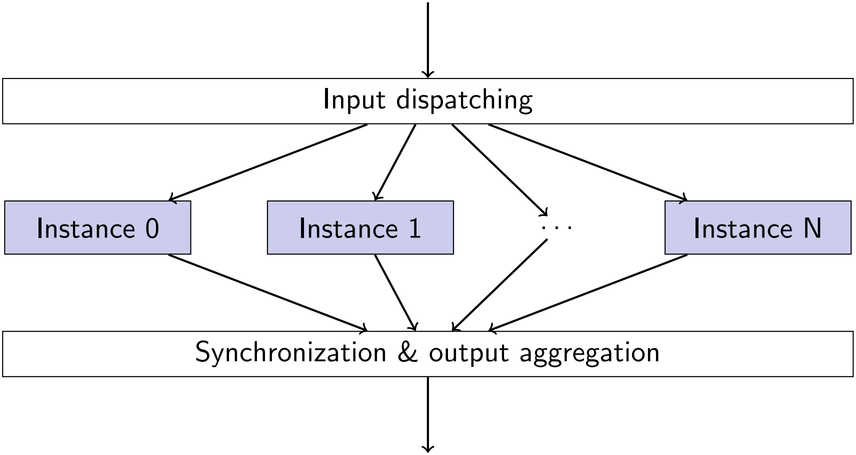
In biology, maintaining high genetic diversity allows species to adapt to future environmental changes, survive from deadly diseases, and avoid inbreeding.

Intuition: gene/DNA diversity

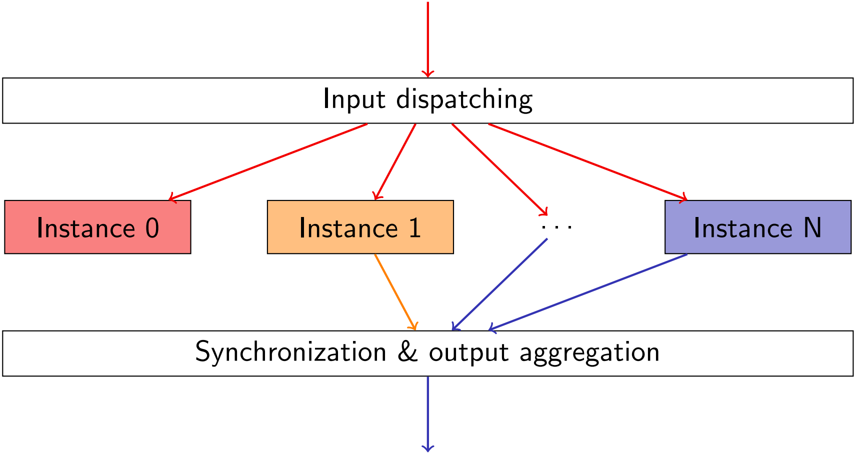
In biology, maintaining high genetic diversity allows species to adapt to future environmental changes, survive from deadly diseases, and avoid inbreeding.

Similarly, we expect software diversity to protect software systems (especially critical systems) from deadly viruses and attacks while also serving as an early signal of being attacked.

Core architecture



Core architecture (under attack)



Challenges of applying diversity-based defenses

- Source of diversity
- Synchronization of diversified instances

Source of diversity

- Compiler/loader-assisted diversity
  - e.g., direction of stack growth
  - e.g., different canary values
  - e.g., different sanitizer instrumentation

Source of diversity

- Compiler/loader-assisted diversity
  - e.g., direction of stack growth
  - e.g., different canary values
  - e.g., different sanitizer instrumentation

- N-version programming
  - e.g., different language VM (V8 vs SpiderMonkey)
  - e.g., different applications (nginx vs apache web server)
  - e.g., similar applications from independent vendors/teams

Source of diversity

- Compiler/loader-assisted diversity
  - e.g., direction of stack growth
  - e.g., different canary values
  - e.g., different sanitizer instrumentation
- N-version programming
  - e.g., different language VM (V8 vs SpiderMonkey)
  - e.g., different applications (nginx vs apache web server)
  - e.g., similar applications from independent vendors/teams
- Platform diversity
  - e.g., different `libc` implementations (glibc vs musl libc)
  - e.g., Adobe Reader on MacOS and Windows
  - e.g., Server programs on Intel and ARM CPUs

Mode of synchronization

- Online mode (via rendezvous points)
- Offline mode (via record-and-replay)

The key is to synchronize all sources of nondeterminism.

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