Changelog

3 May 2021: adjust phrasing about in-edges on 'control flow graph' slide

3 May 2021: CFI prevents exercise: add explanation + E. none of these options

3 May 2021: Android shadow stacks: adjust font size of references

3 May 2021: clang CFI example: correct shown mask to have 5 entries

4 May 2021: CFI prevents exercise: add more explanations

last time

sandboxing integration with UI

- file selection outside sandbox as way to specify accessible files
- Qubes — marking of 'which sandbox' via borders in UI

sandboxing without OS support

- compiling C/C++ code for language VMs
- key insight: keep bounds in sandbox, not in array in sandbox

interfaces for sandboxing?

application permissions

- avoiding wasting user attention
- communicating what permissions do (or not)
- evading permissions via missed interfaces
- evading permissions via cooperation with other apps

a correction

explained Cloak+Dagger incorrectly

one issue: system alert + accessiblity permission = complete control

hide app + interact on user's behalf

another issue: use system alert permission to hide all part of "allow" button

note on CHALLENGE

assignment released

schedule showed wrong due time briefly – 9PM on 12 May (not midnight)

mitigations so far

stopping attacker from writing working code

- write XOR execute
- removing "gadgets"
- address space layout randomization

validating/protecting code pointers

- stack canaries
- full RELRO (protect linker stub pointers, pointers within VTable)
- missing: doing this for other code pointers?
- address space layout randomization

preventing out-of-bounds accesses

- guard pages

mitigation practicality concerns

backwards compatibility

- (high) works with unmodified libraries+programs
- sometimes works with unmodified libraries+program
  - as long as they don't do something 'weird'
- requires recompile of module to be protected
- changes calling conventions: recompile module + things linked with it
- (low) requires changing source code

hardware support required

extra space required

extra time required

other mitigation ideas?

some big categories we haven't seen:

protecting code pointers w/o vulnerability to information leaks

validating/protecting non-return address code pointers

preventing out-of-bounds accesses

intuition: shadow stacks

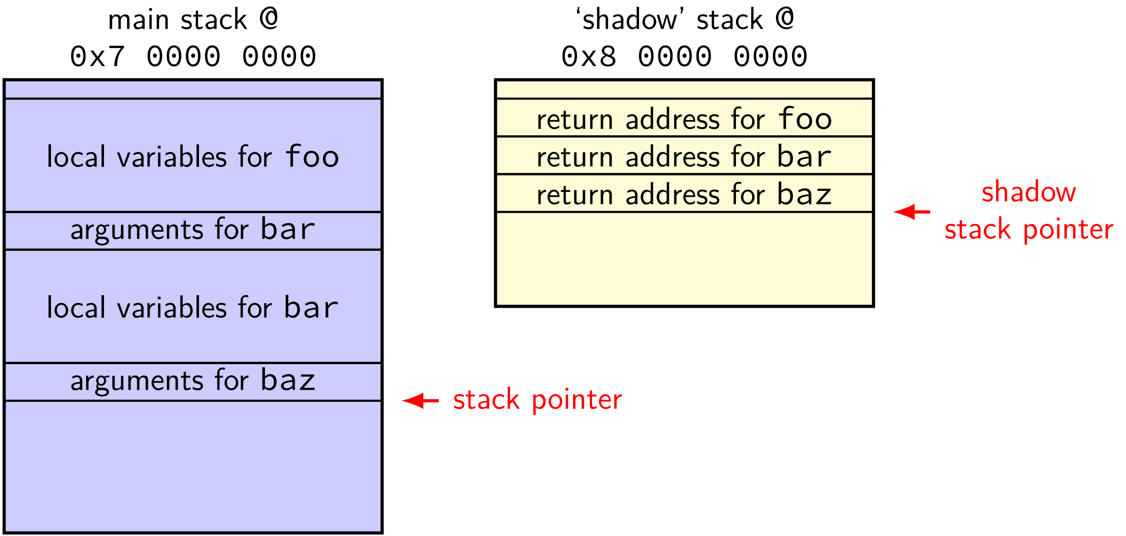
problem with stack: easy to leak address/values because used for lots of data

goal: keep sensitive data in **separate region**

- easier to kepe address secret?

can use this for (stronger?) alternative to stack canaries

shadow stacks



implementing shadow stacks

bigger changes to compiler than canaries

more overhead to call/return from function

most commonly: store return address twice

shadow stacks on x86-64 (1)

idea 1: dedicate %r15 as shadow stack pointer,
copy RA to shadow stack pointer in function prologue

```
function:
    movq (%rsp), %rax      // RAX ← return address
    addq $−8, %r15         // R15 ← R15 - 8
    movq %rax, (%r15)      // M[R15] ← RAX
    ...
    movq (%rsp), %rdx        // RDX ← return address
    cmpq %rdx, (%r15)
    jne CRASH_THE_PROGRAM // if RDX != M[R15] goto C
    add $8, %r15             // R15 ← R15 - 8
    ret
```

shadow stacks on x86-64 (2)

idea 2: dedicate %r15 as shadow stack pointer,
avoid normal call/return instruction

```
    addq $-8, %r15
    leaq after_call(%rip), %rax
    movq %rax, (%r15)
    jmp function
after_call:

function:
    ...
    addq $8, %r15            // R15 ← R15 + 8
    jmp *-8(%r15)            // jmp M[R15-8]
```

Android/AArch64 shadow stacks (1)

via https://clang.llvm.org/docs/ShadowCallStack.html (see also https://security.googleblog.com/2019/10/protecting-against-code-reuse-in-linux_30.html)

dedicate register x18 to shadow stack pointer

x30 = return address (after ARM's call instruction (bl))

ARM call instruction saves return address in register…

without

```
stp     x29, x30, [sp, #-16]!
mov     x29, sp
bl      bar
add     w0, w0, #1
ldp     x29, x30, [sp], #16
ret
```

with shadow stack

```
str     x30, [x18], #8
stp     x29, x30, [sp, #-16]!
mov     x29, sp
bl      bar
add     w0, w0, #1
ldp     x29, x30, [sp], #16
ldr     x30, [x18, #-8]!
ret
```

Intel CET shadow stacks

future Intel processor extension adds shadow stacks

- "Control-flow Enforcement Technology"

new shadow stack pointer

CALL/RET: push/pop from BOTH stacks

shadow stack pages are marked as read-only in page table

- cannot be written through normal instructions
- extra bit identifying as shadow stack (not "normal" read-only page)

preventing shadow stack writes?

ARM64 scheme: prevent writes if

- shadow stack pointer is never leaked (dedicated register)
- shadow stack random location can't be guessed (or queried otherwise)

Intel CET: prevent writes unless

- OS (priviliged/kernel mode) instructions to setup shadow stack used

can we prevent writes without relying on avoiding info leaks…
and without special hardware support?

- well, yes, but …

some early stack canary benchmarks

from Chiueh and Hsu, "RAD: A Compile-Time Solution to Buffer Overflow Attacks" (2001)

Program size	Program tested	User time	System time
11991 lines	Original ctags	0.57	0.05
	MineZone RAD-protected ctags	0.58	0.05
	Read-Only RAD-protected ctags	8.16	19.17

Table 3 Macro-benchmark results of ctags

Program size	Program tested	User time	System time
4500 lines	Original gcc	3.53	0.19
	Mine Zone RAD-protected gcc	4.67	0.2
	Read-Only RAD-protected gcc	20.46	50.43

Table 4 Macro-benchmark results of gcc

automatic shadow stacks?

if we change how CALL/RET works...

...maybe we can add shadow stack support to existing programs?

- either with hardware support, or
- in software with emulation techniques?

well, there's a problem...

the problem in C++

```
void Foo() {
    try {
        ... Bar() ...
    } except (std::runtime_error &error) {
        ...
    }
}

void Bar() {
    ... Quux() ...
}
void Quux() {
    ...
    throw std::runtime_error("...");
    ...
}
```

the problem in C

```
jmp_buf env;
const char *error;
void Foo() {
    if (0 == setjmp(env)) {
        Bar();
    } else {
        ...
    }
}

void Bar() {
    ... Quux() ...
}
void Quux() {
    ...
    error = "...";
    longjmp(env, 1);
    ...
}
```

dealing with non-local returns

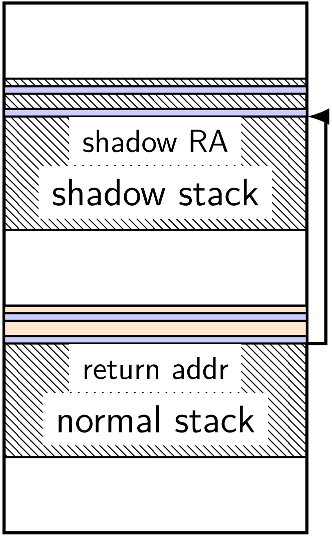
exceptions and setjmp/longjmp deliberately skip return calls

one solution: “direct” shadow stack

fixed (possibly secret) offset from normal stack

shadow stack only stores return addreses

- space in between return addresses left as nulls



what do shadow stacks stop?

combined with a information leak that can dump arbitrary bytes of memory,
which of these exploits would shadow stacks stop...

A. using format string exploit to point stack return address to the ‘system’ function
B. using format string exploit to point VTable to the ‘system’ function
C. using an unchecked string copy that goes over the end of a stack buffer into the return address and pointing the return address to the ‘system’ function
D. using a buffer overflow that overwrites a saved stack pointer value to cause return to use a different address
E. using pointer subterfuge to overwrite the GOT entry for ‘printf’ to point to the ‘system’ function

a simple way to check returns?

observation: places we return to usually after call instructions

exception: 'tail calls' — we'll ignore this for now

we could check for one...

replace return with:

return address ← PopFromStack()
if DecodeInstruction(return address - 5) == "call thisFunction":
    goto return address
else:
    CRASH

a simple way to check returns?

more practical: `label $ID` instruction with encoding:

TWO-BYTE-OPCODE FOUR-BYTE-CONSTANT

(real version: can reuse some sufficiently nop-like instruction)

```
...
    call foo
    label $0xf19279bb // random ID for function foo
...
```

```
foo:
...
    pop %rdx              // RDX ← return address
    cmp $0xf19279bb, 2(%rdx)
    jne CRASH
    jmp *%rdx
```

looks like canaries? (1)

what attacks does this stop that canaries don't?

looks like canaries? (1)

what attacks does this stop that canaries don't?

ID does not need to be secret!

- assuming non-executable writeable memory, no!
- attacker can't write new places for return to go

looks like canaries? (1)

what attacks does this stop that canaries don't?

ID does not need to be secret!

- assuming non-executable writeable memory, no!
- attacker can't write new places for return to go

avoids "stack pivoting" attacks

- attacker can't make stack pointer point to wrong part of stack...
- and expect it to return differently

looks like canaries? (2)

what attacks does this NOT stop that canaries don't?

example: SortList can be called from `Innocent`,
then return from `Dangerous`

> assumption: attacker can overwrite return address at right time (running on another core? problem with sortFunc1?)

```
void Innocent() {
  ...
  SortList(someList1,
           sortFunc1);
  Use(someList1);
  ...
}
```

```
void Dangerous() {
  ...
  SortList(someList2,
           sortFunc2);
  UseDangerously(someList2);
  ...
}
```

checking a VTable call

```
class A { public:
  virtual void bar() { ... }
};
class B : public A { public:
  void bar() { ... }
};
void example(A *obj) {
  obj->bar();
}
```

```
example:
  // rax ← vtable address
  movq (%rdi), %rax
  // rdx ← first vtable entry
  movq (%rax), %rax
  // call using vtable entry
  call *%rax
  ...
```

checking a VTable call

```
class A { public:
  virtual void bar() { ... }
};
class B : public A { public:
  void bar() { ... }
};
void example(A *obj) {
  obj->bar();
}
```

```
example:
  // rax ← vtable address
  movq (%rdi), %rax
  // rdx ← first vtable entry
  movq (%rax), %rax
  // call using vtable entry
  call *%rax
  ...
```

example uses VTable to call method
target for memory corruption attacks
just like return addresses
so, apply same strategy

checking a VTable call

```
class A { public:
  virtual void bar() { ... }
};
class B : public A { public:
  void bar() { ... }
};
void example(A *obj) {
  obj->bar();
}
```

```
A::bar():
  label $0xe0c5df0b
  ...
B::bar():
  label $0xe0c5df0b
  ...
```

```
example:
  // rax ← vtable address
  movq (%rdi), %rax
  // rdx ← first vtable entry
  movq (%rax), %rax
  // call using vtable entry
  call *%rax
  ...
```

```
example:
  movq (%rdi), %rax
  movq (%rax), %rax
  cmpq $0xe0c5df0b, 2(%rax)
  jne CRASH
  call *%rax
  ...
```

checking a VTable return

```
A::bar():
   label $0xe0c5df0b
   ...
   pop %rdx // RDX ← return address
   cmp $0x64a0cfe3, 2(%rdx)
   jne CRASH
   jmp *%rdx
B::bar():
   label $0xe0c5df0b
   ...
   pop %rdx // RDX ← return address
   cmp $0x64a0cfe3, 2(%rdx)
   jne CRASH
   jmp *%rdx
```

```
example:
   movq (%rdi), %rax
   movq (%rax), %rax
   cmpq $0xe0c5df0b, 2(%rax)
   jne CRASH
   call *%rax
   label $0x64a0cfe3
   ret
```

if we want to use this label-checking on the return
need to choose the same label for A::bar and B::bar return, too

calls through function pointers

```
typedef int (*CompareFnType)(const char*, const char*)
void SortFunction(const char **items, CompareFnType compare) {
    ...
    (*compare)(a, b);
    ...
}
```

here: call through explicitly passed function pointer

want to do the same thing we did for VTable calls

- all the compare functions have the same label
- all the returns form compare functions have the same label

yes, if we can somehow label all the compare functions

calls through function pointers

```
typedef int (*CompareFnType)(const char*, const char*)
void SortFunction(const char **items, CompareFnType compare) {
     ...
     (*compare)(a, b);
     ...
}
```

here: call through explicitly passed function pointer

want to do the same thing we did for VTable calls

- all the compare functions have the same label
- all the returns form compare functions have the same label

yes, if we can somehow label all the compare functions

CFI overhead

Abadi et al's 2004 paper:

- used label-based approach
- 0-45% time overhead on SPECcpu2000 benchmarks
- best: compression program
- worst: chess engine

Tice et al's 2014 paper (clang-style impl, sometimes in GCC, sometimes in Clang)

- could seperately enable different parts
- in tests on SPECcpu 2006 benchmarks:
- 0–10% slowdown for VTable dereference checks
  - but 20% without tuning
- 0-6% for other indirect call checking

looks like canaries? (2)

what attacks does this NOT stop that canaries don't?

example: SortList can be called from `Innocent`,
then return from `Dangerous`

- assumption: attacker can overwrite return address at right time (running on another core? problem with sortFunc1?)

```
void Innocent() {
  ...
  SortList(someList1,
           sortFunc1);
  Use(someList1);
  ...
}
```

```
void Dangerous() {
  ...
  SortList(someList2,
           sortFunc2);
  UseDangerously(someList2);
  ...
}
```

concept: labels and control flow graph

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```

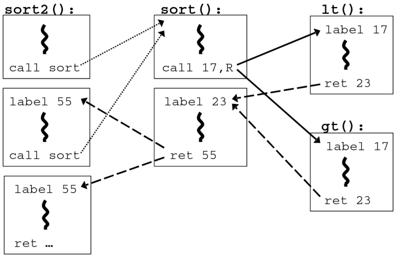


Figure 1: Example program fragment and an outline of its CFG and CFI instrumentation.

control flow graph

nodes = blocks of code

edges = *potential jump/call*

assigning labels: every in-edge needs to check same label at source

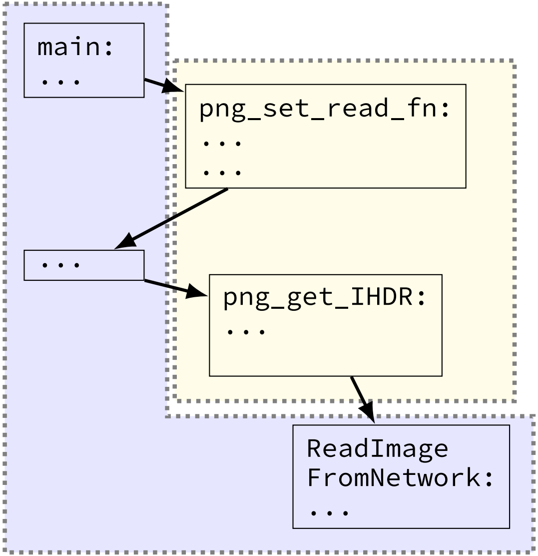
figure from Abadi et al, "Control-Flow Integrity: Principles, Implementations and Applications" (CCS 2005)

library-crossing CFGs

main.c

```
#include <png.h>
void ReadImageFromNetwork(
    png_structp libpng_handle,
    unsigned char *bytes,
    size_t size
) { ...  }

int main() {
    /* init libpng */
    png_structp libpng_handle = ...;
    /* tell libpng how to read image data */
    png_set_read_fn(
        libpng_handle, ...,
        ReadImageFromNetwork
    )
    ...
    /* extract "header"
       information from image */
    png_get_IHDR(libpng_handle, ...)
    ...
}
```



CFGs will be imprecise

```
FunctionPtr p = functionA;
Example() {
  while (true) {
    ...
    if (SomethingComplicated()) {
      (*p)();
    } else if (SomethngElseComplicated()) {
      foo();
    }
    ...
  }
}
foo() {
  ...
  if (AnotherComplexThing()) {
    p = functionB;
  }
}
```

can Example() call functionB()? probably not practical to tell

need to make conservative 'yes' guess

finding possible function pointer values?

given call using function pointers
how do we find the **legitimate** possible values?

one high-level idea:

```
for each fptr constant X:
    PossibleValues[X] = {X}
for each fptr variable X:
    PossibleValues[X] = empty set
until PossibleValues stops changing:
    for each fptr assignment LHS=RHS:
        for each fptr variable/constant Y
                that RHS could evaluate to:
            PossibleValues[LHS] = Union(
                PossibleValues[LHS],
                PossibleValues[Y]
            )
```

...but not so easy to ID function pointer vars/assignments?

finding possible function pointer values?

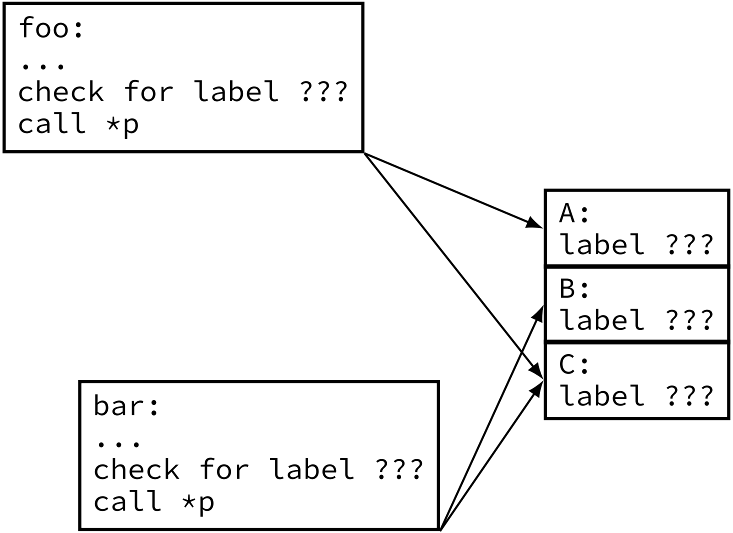
given call using function pointers
how do we find the **legitimate** possible values?

one high-level idea:

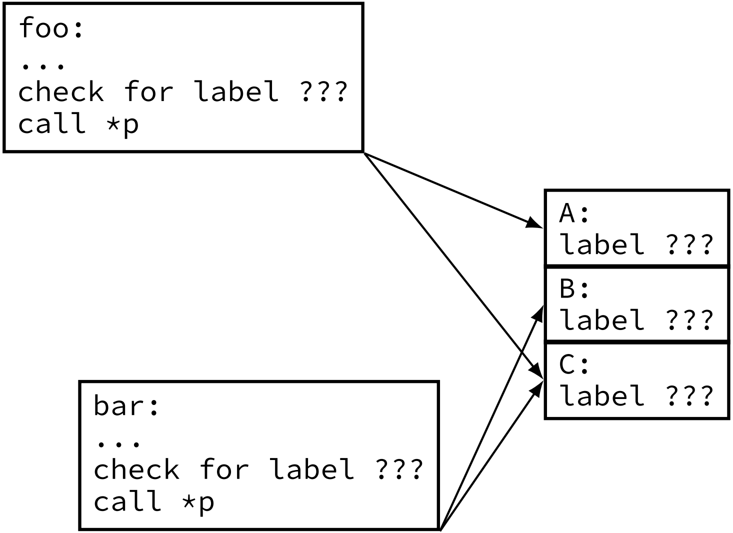
```
for each fptr constant X:
    PossibleValues[X] = {X}
for each fptr variable X:
    PossibleValues[X] = empty set
until PossibleValues stops changing:
    for each fptr assignment LHS=RHS:
        for each fptr variable/constant Y
                that RHS could evaluate to:
            PossibleValues[LHS] = Union(
                PossibleValues[LHS],
                PossibleValues[Y]
            )
```

...but not so easy to ID function pointer vars/assignments?

labels aren't enough?



labels aren't enough?



two possible fixes:

make checks scan for multiple labels (more overhead)

allow foo to call B and bar to call A (easier to attack)

clang's CFI implementation

`https://clang.llvm.org/docs/ControlFlowIntegrity.html`

- also `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice`

only checks calls via VTables or function pointers

stable implementation requires libraries compiled with support

label information is placed in separate data structure

- looked up using function or VTable addresses

trick: keep functions in one region of memory

clang's CFI implementation

`https://clang.llvm.org/docs/ControlFlowIntegrity.html`

also `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice`

only checks calls via VTables or function pointers

stable implementation requires libraries compiled with support

label information is placed in separate data structure

looked up using function or VTable addresses

trick: keep functions in one region of memory

clang idea for CFI indirect calls

```
start_funcs_with_two_string_args:
.align 8
compare_alpha:
  jmp real_compare_alpha
.align 8
run_command_with_arg:
  jmp real_run_command_with_arg
.align 8
print_two_strings:
  jmp real_print_two_strings
.align 8
move_file:
  jmp real_move_file
.align 8
compare_reverse_alpha:
  jmp real_compare_reverse_alpha
end_funcs_with_two_sting_args:
```

functions of same type placed together

every func's address is multiple of 8

clang idea for CFI indirect calls

```
start_funcs_with_two_string_args:
.align 8
compare_alpha:
  jmp real_compare_alpha
.align 8
run_command_with_arg:
  jmp real_run_command_with_arg
.align 8
print_two_strings:
  jmp real_print_two_strings
.align 8
move_file:
  jmp real_move_file
.align 8
compare_reverse_alpha:
  jmp real_compare_reverse_alpha
end_funcs_with_two_sting_args:
```

check pseudocode:
round fptr to multiple of 8
if fptr $<$ start or fptr $>$ end:
    CRASH
allowed $\leftarrow$ [1,0,0,0,1]
    'mask' for compare funcs
offset $\leftarrow$ fptr - start
if bit (offset/8) of allowed
        is not set:
    CRASH

clang idea for VTables

check VTable element address instead of function address

otherwise

- place all VTables for related classes together
- check start/end address for VTables
- bit mask indicating which VTable entries are okay for call

CFI prevents?

```
class Foo { public:
    virtual void f() { }
};
class Bar : public Foo { public:
    virtual void f() { g(1); }
};
class Quux : public Foo { public:
    virtual void f() { }
};
void g(int x) {
    if (x == 0) { danger(); }
}
int h(int x) { return 0; }
int (*ptr)(int) = &h;
```

with clang's CFI, which likely can end up calling `danger()` if an attacker can first write to arbitrary memory locations?

A. `(*ptr)(1);`
B. `(*ptr)(0);`
C. `Foo *q = attacker_controlled(); q->f()`
D. `Quux *q = attacker_controlled(); q->f()`
E. none of these

CFI prevents?

```
class Foo { public:
    virtual void f() { }
};
class Bar : public Foo { public:
    virtual void f() { g(1); }
};
class Quux : public Foo { public:
    virtual void f() { }
};
void g(int x) {
    if (x == 0) { danger(); }
}
int h(int x) { return 0; }
int (*ptr)(int) = &h;
```

with clang's CFI, which likely can end up calling `danger()` if an attacker can first write to arbitrary memory locations?

A. `(*ptr)(1);`

B. `(*ptr)(0);` if compiler thinks ptr set to g ever, yes; otherwise, no

C. `Foo *q = attacker_controlled(); q->f()` can only call real f() methods; could call Bar::f() but how to change g's arg?

D. `Quux *q = attacker_controlled(); q->f()` can only

backup slides

pointers to function pointers?

```
struct contains_fptr {
    char buffer[1024];
    func_ptr_t func_ptr;
};

struct contains_fptr x, y;
struct contains_fptr *p;
p = &y;
...
if (Q) {
    p = &x; // this effectively changes function pointer!
}
```

typical solution: try to track everything *every pointer* can point to

similar algorithm (sets of possible values for each pointer)

“aliasing analysis”

arrays of function pointers?

```
func_ptr_t arr[1024];

arr[X] = p;
q = arr[Y];
```

what if we don't know X, Y

one idea: assume assignment to/from each possible array element

worth the effort?

isn't all this analysis really tricky?

yes, but...compilers want/try to do this for other reasons

very useful to know possible pointer values for optimizations

very useful to know what a function call can do for optimizations

CFI: dynamically loaded libraries?

what about dynamically loaded libraries…

problem: precomputed control flow graph now invalid

Intel hardware CFI support

Intel adds ‘endbr64’ instruction

special NOP instruction that acts as a label

means: only one label for everything

prevents gadgets from existing

“Control Flow Enforcement”: if enabled
computed jump to non-endbr64 triggers segfault-like error

ARM has similar feature called Branch Target Identification

authenticated pointers (1)

```
some_function:
    authentication_code <— hash(
        secret key,
        return address
    )
    ... dangerous function code ...
    assert(authenication_code ==
        hash(secret key, return addres))
    jump to return address
```

authenticated pointers (2)

```
some_function:
    authentication_code <— hash(
        secret key,
        stack pointer,
        return address
    )
    ... dangerous function code ...
    assert(authenication_code ==
        hash(secret key, stack pointer, return address))
    jump to return address
```

authenticated pointers (3)

```
some_function:
    return address <— encode(
        secret key,
        stack pointer,
        return address
    )
    ... dangerous function code ...
    return address <— decode_or_crash(
        secret key,
        stack pointer,
        return address
    )
    jump to return address
```

authenticated pointers (4)

```
some_vtable[index] <— encode(
    secret key,
    label,
    address of some of function
)
... dangerous code ...
function pointer <— decode(
    secret key,
    label,
    object—>vtable[index]
)
call function pointer
```

ARM authenticated pointers

ARM64 implements this idea with:

secret key kept in a special register (hard to leak to attacker)

authentication code placed in upper pointer bits

- makes pointer temporarily invalid
- can't "accidentally" use authenticated pointer without verifying authentication code first

authenticated pointer layout

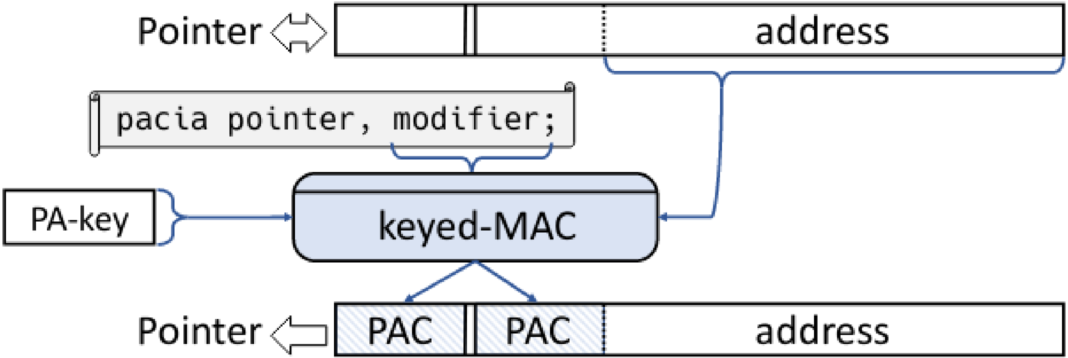


Figure 1: The PAC is created using key-specific PA instructions (`pacia`) and is a keyed MAC calculated over the pointer address and a modifier.

authentication keys

processes can have multiple authentication keys active

easy to use separate keys for

- return address pointers
- function pointers
- any pointers to data

authentication keys are in special registers — need OS to read/set

also can "mix" in extra info like stack pointer