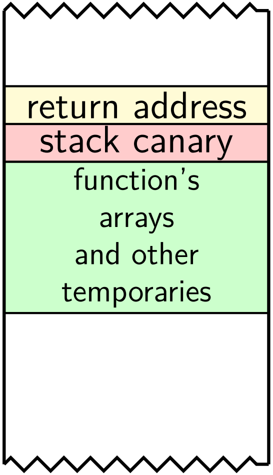
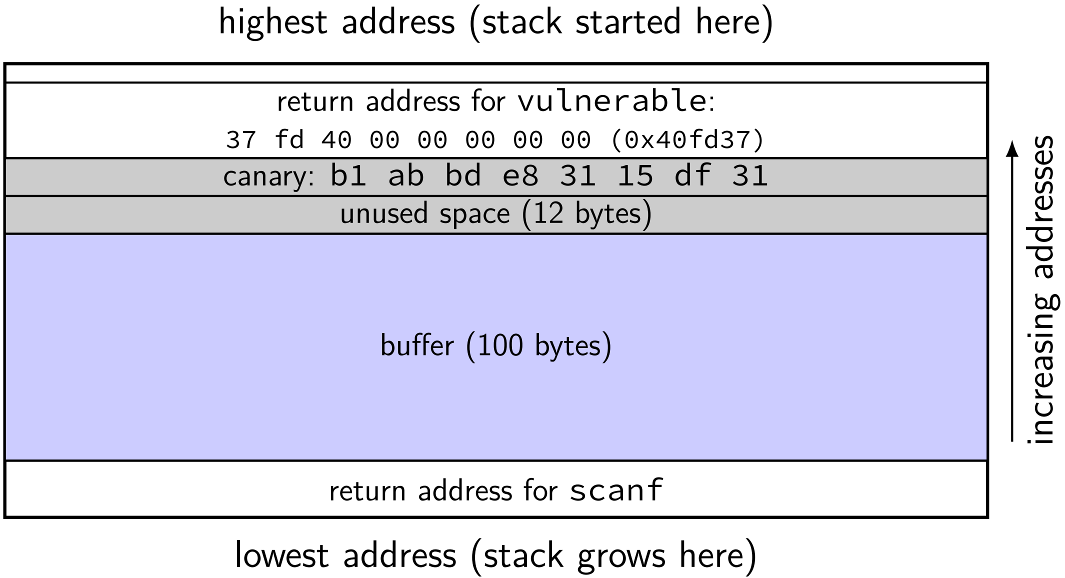
compiler generated code

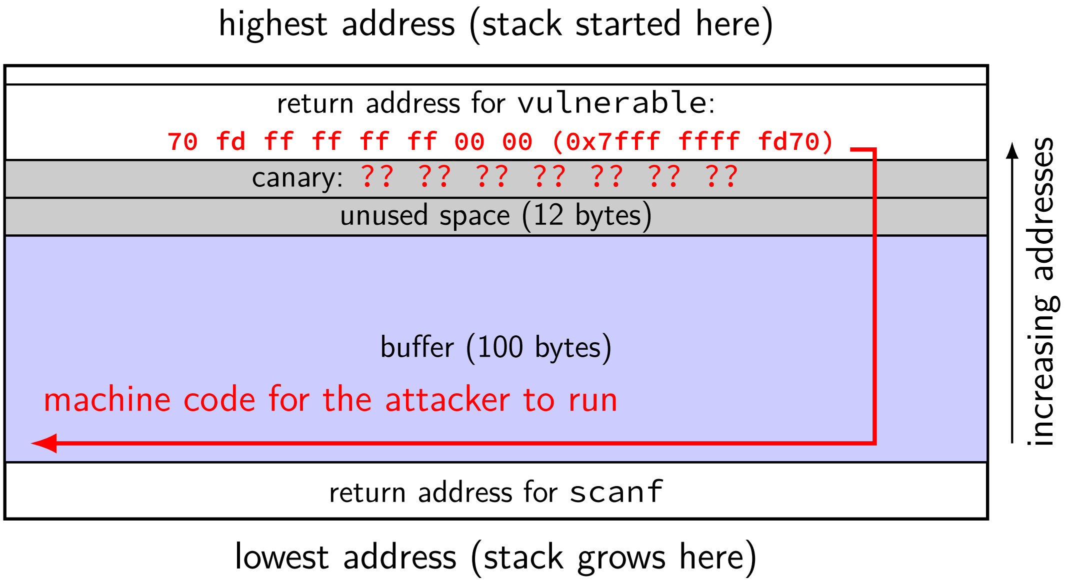
```
    pushq %rbx
    sub $0x20,%rsp
/* copy value from thread-local storage */
    mov $0x28,%ebx
    mov %fs:(%rbx),%rax
/* onto the stack */
    mov %rax,0x18(%rsp)
/* clear register holding value */
    xor %eax, %eax
    ...
    ...
/* copy value back from stack */
    mov 0x18(%rsp),%rax
/* xor to compare */
    xor %fs:(%rbx),%rax
/* if result non-zero, do not return */
    jne call_stack_chk_fail
    ret
call_stack_chk_fail:
```



stack canary



stack canary



stack canary hopes

overwrite return address $\implies$ overwrite canary

canary is secret

good choices of canary

random — guessing should not be practical

- not always — sometimes static or only 2^{15} possible

GNU libc: canary contains:

leading `\0` (string terminator)

- `printf %s` won't print it
- copying a C-style string won't write it

a newline

- read line functions can't input it

`\xFF`

- hard to input?

stack canaries implementation

"StackGuard" — 1998 paper proposing strategy

GCC: command-line options

- `-fstack-protector`
- `-fstack-protector-strong`
- `-fstack-protector-all`
- one of these often default
- three differ in how many functions are 'protected'

Microsoft C/C++ compiler: `/GS`

- on by default

stack canary overheads

less than 1% runtime if added to "risky" functions

- functions with character arrays, etc.

large overhead if added to all functions

- StackGuard paper: 5–20%?

similar space overheads

(for typical applications)

- could be much worse: tons of 'risky' function calls

stack canaries pro/con

pro: no change to calling convention

pro: recompile only — no extra work

con: can't protect existing executable/library files (without recompile)

con: doesn't protect against many ways of exploiting buffer overflows

con: vulnerable to information leaks

stack canaries pro/con

pro: no change to calling convention

pro: recompile only — no extra work

con: can't protect existing executable/library files (without recompile)

con: *doesn't protect against many ways of exploiting buffer overflows*

con: vulnerable to information leaks

stack canaries pro/con

pro: no change to calling convention

pro: recompile only — no extra work

con: can't protect existing executable/library files (without recompile)

con: doesn't protect against many ways of exploiting buffer overflows

con: *vulnerable to information leaks*

stack canary summary

stack canary — simplest of many *mitigations*

key idea: detect corruption of return address

assumption: if return address changed, so is adjacent token

assumption: attacker can't learn true value of token

- often possible with memory bug

later: workarounds to break these assumptions

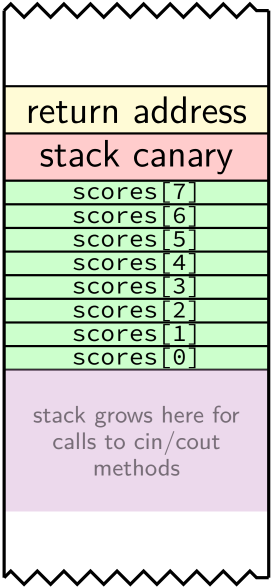
stack canary hopes

overwrite return address $\implies$ *overwrite canary*

canary is secret

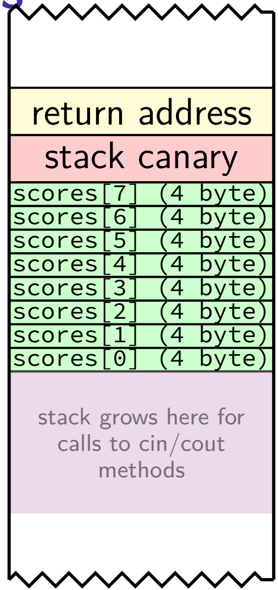
non-contiguous overwrites

```
void vulnerable() {
  int scores[8]; bool done = false;
  while (!done) {
    prinf("Edit␣which␣score?␣(0␣to␣7)␣");
    int i;
    scanf("%d\n", &i);
    /* Oops!
       sizeof(scores) is 4 * sizeof(int) */
    if (i < 0 || i >= sizeof(scores))
      continue;
    printf("Set␣to␣what␣value?␣");
    scanf("%d", &scores[i]);
    ...
  }
  ...
}
```



exercise: non-contiguous overwrites

```
void vulnerable() {
  int scores[8]; bool done = false;
  while (!done) {
    prinf("Edit␣which␣score?␣(0␣to␣7)␣");
    int i;
    scanf("%d\n", &i);
    /* Oops!
       sizeof(scores) is 4 * sizeof(int) */
    if (i < 0 || i >= sizeof(scores))
      continue;
    printf("Set␣to␣what␣value?␣");
    scanf("%d", &scores[i]);
```



exercise: to set return address to 0x123456789, set what scores to what values?

```
}
```

```
0x123456789
0x0000 0001 2345 6789
89 67 45 23 01 00 00 00
[89 67 45 23] [01 00 00 00]
0x2345678 0x1
```

stack canary hopes

overwrite return address $\implies$ overwrite canary

canary is secret

information disclosure (1a)

```
void vulnerable() {
    int value;
    for (;;) {
        command = ReadInput();
        if (command == "set") {
            value = ReadIntInput();
        } else if (command == "get") {
            printf("%d\n", value);
        } else if ...
    }
}
```

"get" command: can read *uninitialized value*

example: when I compiled this, `value` was stored on the stack

information disclosure (1b)

```
void vulnerable() {
    int value;
    ...
        } else if (command == "get") {
            printf("%d\n", value);
        }
    ...
}
void leak() {
    int secrets[] = {
        12345678, 23456789, 34567890,
        45678901, 56789012, 67890123,
    };
    do_something_with(secrets);
}
int main() {leak(); vulnerable();}
```

running this program (input in bold):

get

67890123

information disclosure (2)

```
void process() {
    char buffer[8] = "\0\0\0\0\0\0\0\0";
    char c = '␣';
    for (int i = 0; c != '\n' && i < 8; ++i) {
        c = getchar();
        buffer[i] = c;
    }
    printf("You␣input␣%s\n", buffer);
}
```

input aaaaaaaa

output `You input aaaaaaaa`*(whatever was on stack)*

information disclosure (3)

```
struct foo {
    char buffer[8];
    long *numbers;
};

void process(struct foo* thing) {
    ...
    scanf("%s", thing->buffer);
    ...
    printf("first␣number:␣%ld\n", thing->numbers[0]);
}
```

input: aaaaaaaa*(address of canary)*

address on stack *or* where canary is read from in thread-local storage

repeated reads

sometimes find “read gadgets”

- example buffer overflow into pointer

often reusable (e.g. input in loop in server)

can find value with multiple steps

- read global pointer that points in middle of array on stack, then
- then read that pointer + 8, pointer + 16, etc. until finding stack canary

can leak 8+ bytes with repeated 1-byte leak

exercise (1)

```
struct point {          struct point p;
    int x, y, z;;       ...
};                          if (command == "get") {
                                /* 'p' could be uninitialized */
                                printf("%d,%d,%d\n", p.x, p.y, p.z);
                            } ...
                        ...
```

Suppose p ("left over" from prior use of register, etc.) is stored at the same address of an 'leftover' copy of the 8-byte stack canary. If 999999,44444,333333 is output, how do we compute the stack canary value?

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	Real time
11991 lines	Original ctags	0.57	0.05	0.62
	MineZone RAD-protected ctags	0.58	0.05	0.63
	Read-Only RAD-protected ctags	8.16	19.17	27.32

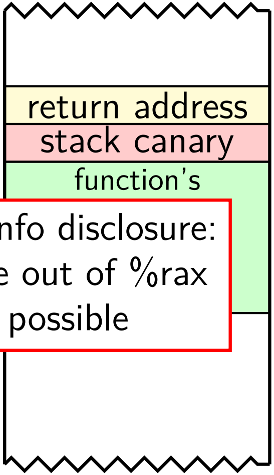
Table 3 Macro-benchmark results of ctags

Program size	Program tested	User time	System time	Real time
4500 lines	Original gcc	3.53	0.19	3.72
	Mine Zone RAD-protected gcc	4.67	0.2	4.87
	Read-Only RAD-protected gcc	20.46	50.43	70.89

Table 4 Macro-benchmark results of gcc

compiler generated code

```
    pushq %rbx
    sub $0x20,%rsp
/* copy value from thread-local storage */
    mov $0x28,%ebx
    mov %fs:(%rbx),%rax
/* onto the stack */
    mov %rax,0x18(%rsp)
/* clear register holding value */
    xor %eax, %eax
    ...
    ...
/* copy value back from stack */
    mov 0x18(%rsp),%rax
/* xor to compare */
    xor %fs:(%rbx),%rax
/* if result non-zero, do not return */
    jne call_stack_chk_fail
    ret
call_stack_chk_fail:
```



trying to avoid info disclosure: get canary value out of %rax as soon as possible

intuition: shadow stacks

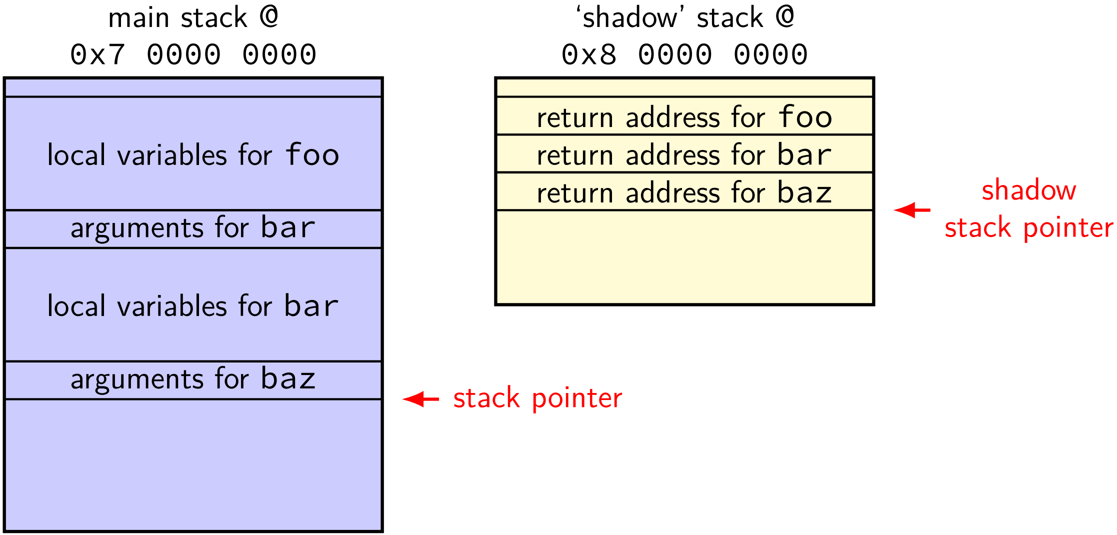
problem with stack: easy to leak address/values because used for lots of data

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
    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
```

Android/AArch64 shadow stacks (2)

`-fsanitize=shadowcallstack`

supported on 64-bit ARM and RISC V only

"An x86_64 implementation was evaluated using Chromium and was found to have critical performance and security deficiencies"

Intel CET shadow stacks

recent Intel processor extension adds shadow stacks

- "Control-flow Enforcement Technology"

new shadow stack pointer

CALL/RET: push/pop from BOTH stacks

shadow stack also protected from writes by hardware + OS

- cannot be written through normal instructions
- modification to page table structures

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);
    ...
```

shadow stacks and non-lcoal returns

need to modify these functions to support shadow stacks, it seems?

violates idea of hardware extension that modifies CALL/RET operation

one way: dealing with non-local returns

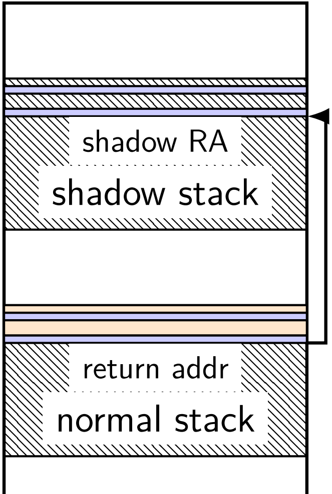
exceptions and setjmp/longjmp deliberately skip return calls

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



CET and shadow-stack manipulation

Intel CET has instructions to manipulate shadow stack pointer

RDSSP (read shadow stck pointer)

- used by glibc setjmp

INCSSP (increment shadow stack pointer)

- apparently used by glibc longjmp in common case

also some functionality for switching shadow stacks

backup slides

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 ...

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