esse

Advanced Security for Systems Engineering – VO 05: Advanced Attacks on Applications 3

Clemens Hlauschek, Daniel Marth, Christian Brem

INSO – Industrial Software
Institute of Information Systems Engineering | Faculty of Informatics | TU Wien

Agenda

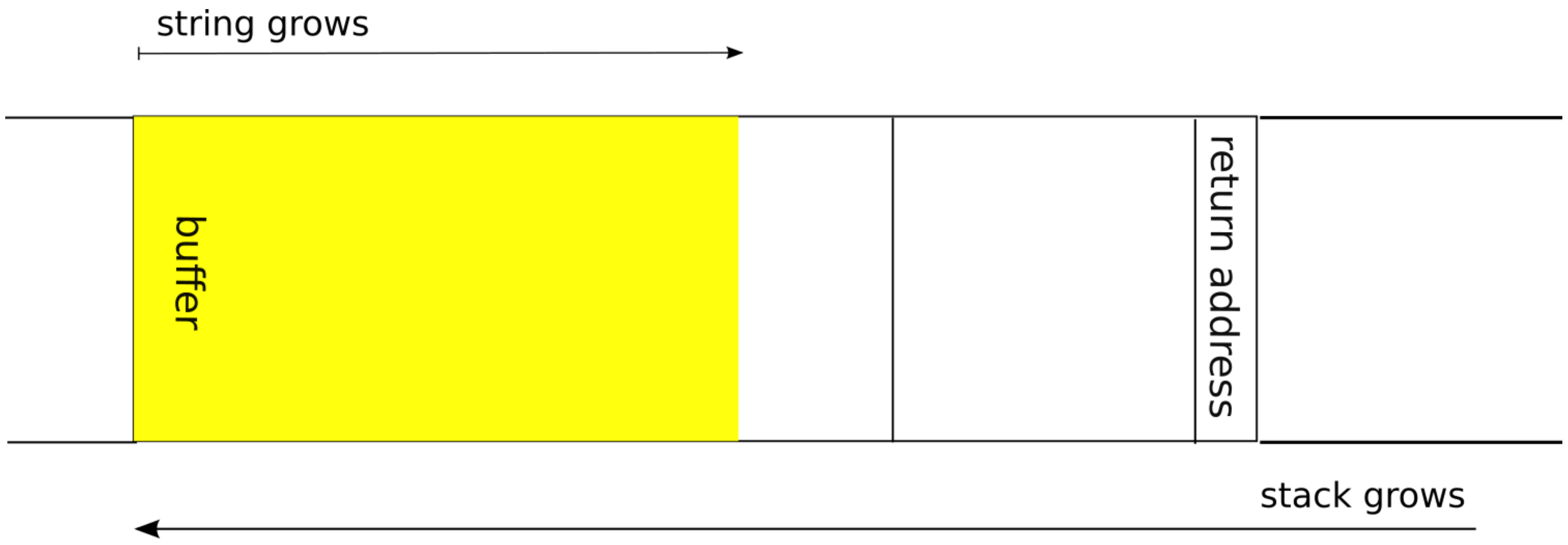
Capture-the-Flag Team defragmented.brains

- Take part in many international hacking competitions
- Diverse bunch, different skills and skill levels
- Join our mailinglist: `ctf-join@inso.tuwien.ac.at`
- Next CTF: HITCON 25.-27.11.

Stack Smashing and Shell Code writing

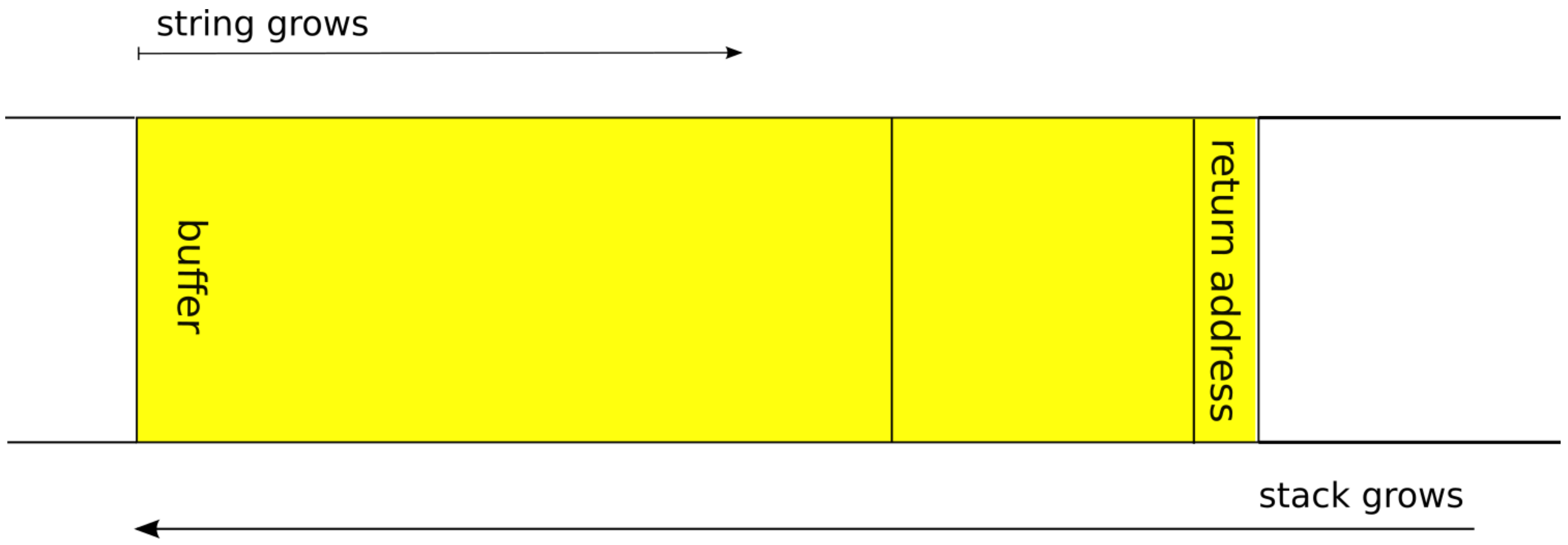
Stack Buffer Overflow: Recapitulation

Basic stack layout, a horizontal perspective



Stack Buffer Overflow: Recapitulation

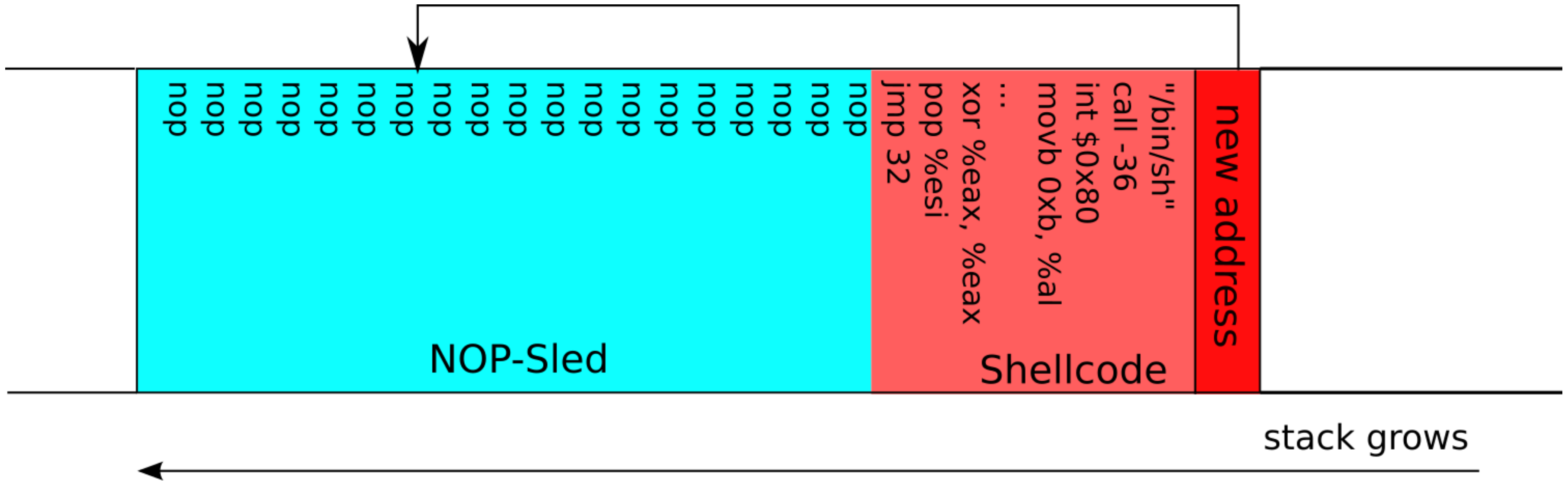
String spills out of buffer, overwrites saved return address.



Stack Buffer Overflow: NOP-Sled

New return address needs to point to buffer: Exact location not known.

- Prepend NOP-Sled to shellcode as "landing zone"
- Make an educated guess for an address somewhere in the NOP-Sled



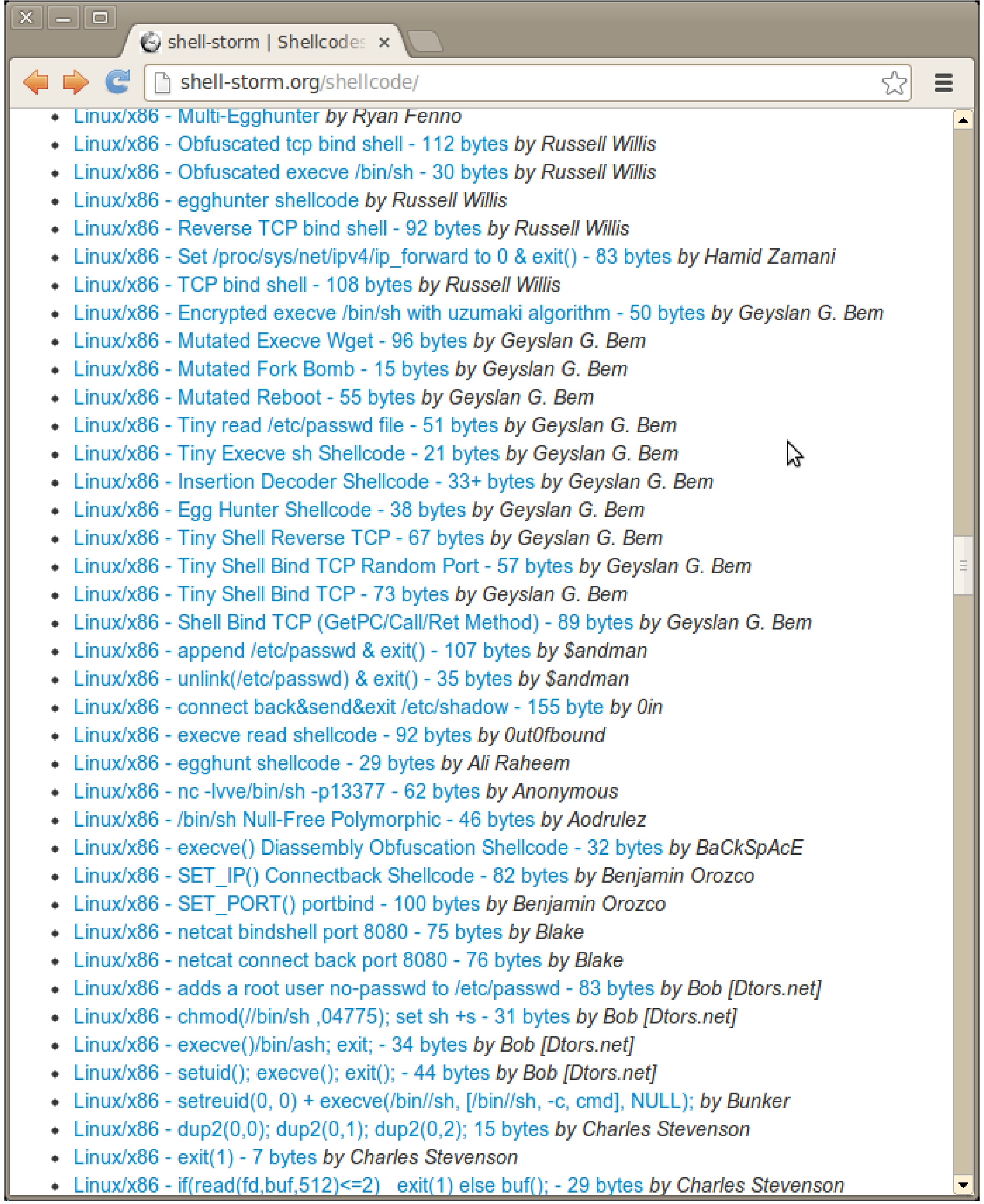
Shellcode

It is called shellcode even if it does not spawn a shell.

- Can do any arbitrary computation
- Useful for an attacker:
  - Bind a shell to a network port
  - Connect back to an attacker
  - Load a post-exploitation framework
  - Start automated malware infection
- A tiny, space-constrained shellcode can be used to load a more powerful “second stage”

Shellcode Writing

- Many different shellcodes available
- For successfull exploitation, it is often necessary to be able to write, debug, and analyze shellcode
- Best to write in assembly



- Linux/x86 - Multi-Egghunter by Ryan Fenno
- Linux/x86 - Obfuscated tcp bind shell - 112 bytes by Russell Willis
- Linux/x86 - Obfuscated execve /bin/sh - 30 bytes by Russell Willis
- Linux/x86 - egghunter shellcode by Russell Willis
- Linux/x86 - Reverse TCP bind shell - 92 bytes by Russell Willis
- Linux/x86 - Set /proc/sys/net/ipv4/ip_forward to 0 & exit() - 83 bytes by Hamid Zamani
- Linux/x86 - TCP bind shell - 108 bytes by Russell Willis
- Linux/x86 - Encrypted execve /bin/sh with uzumaki algorithm - 50 bytes by Geyslan G. Bem
- Linux/x86 - Mutated Execve Wget - 96 bytes by Geyslan G. Bem
- Linux/x86 - Mutated Fork Bomb - 15 bytes by Geyslan G. Bem
- Linux/x86 - Mutated Reboot - 55 bytes by Geyslan G. Bem
- Linux/x86 - Tiny read /etc/passwd file - 51 bytes by Geyslan G. Bem
- Linux/x86 - Tiny Execve sh Shellcode - 21 bytes by Geyslan G. Bem
- Linux/x86 - Insertion Decoder Shellcode - 33+ bytes by Geyslan G. Bem
- Linux/x86 - Egg Hunter Shellcode - 38 bytes by Geyslan G. Bem
- Linux/x86 - Tiny Shell Reverse TCP - 67 bytes by Geyslan G. Bem
- Linux/x86 - Tiny Shell Bind TCP Random Port - 57 bytes by Geyslan G. Bem
- Linux/x86 - Tiny Shell Bind TCP - 73 bytes by Geyslan G. Bem
- Linux/x86 - Shell Bind TCP (GetPC/Call/Ret Method) - 89 bytes by Geyslan G. Bem
- Linux/x86 - append /etc/passwd & exit() - 107 bytes by $andman
- Linux/x86 - unlink(/etc/passwd) & exit() - 35 bytes by $andman
- Linux/x86 - connect back&send&exit /etc/shadow - 155 byte by 0in
- Linux/x86 - execve read shellcode - 92 bytes by 0ut0fbound
- Linux/x86 - egghunt shellcode - 29 bytes by Ali Raheem
- Linux/x86 - nc -lvve/bin/sh -p13377 - 62 bytes by Anonymous
- Linux/x86 - /bin/sh Null-Free Polymorphic - 46 bytes by Aodrulez
- Linux/x86 - execve() Diassembly Obfuscation Shellcode - 32 bytes by BaCkSpAcE
- Linux/x86 - SET_IP() Connectback Shellcode - 82 bytes by Benjamin Orozco
- Linux/x86 - SET_PORT() portbind - 100 bytes by Benjamin Orozco
- Linux/x86 - netcat bindshell port 8080 - 75 bytes by Blake
- Linux/x86 - netcat connect back port 8080 - 76 bytes by Blake
- Linux/x86 - adds a root user no-passwd to /etc/passwd - 83 bytes by Bob [Dtors.net]
- Linux/x86 - chmod(//bin/sh ,04775); set sh +s - 31 bytes by Bob [Dtors.net]
- Linux/x86 - execve()/bin/ash; exit; - 34 bytes by Bob [Dtors.net]
- Linux/x86 - setuid(); execve(); exit(); - 44 bytes by Bob [Dtors.net]
- Linux/x86 - setreuid(0, 0) + execve(/bin//sh, [/bin//sh, -c, cmd], NULL); by Bunker
- Linux/x86 - dup2(0,0); dup2(0,1); dup2(0,2); 15 bytes by Charles Stevenson
- Linux/x86 - exit(1) - 7 bytes by Charles Stevenson
- Linux/x86 - if(read(fd,buf,512)<=2) _exit(1) else buf(); - 29 bytes by Charles Stevenson

Shellcode Writing: Challenges

Special challenges when executing on an indeterminate memory location

- `push` operation can overwrite your shellcode
  - contingently adjust %esp register
- Often, shellcode has to survive `strcpy`, etc
  - No null chars, alphanumeric, upper case shellcode, etc

During normal program building (and loading), the linker adjusts addresses

- String parameters delivered with the payload
- But shellcode does not know its address

Shellcode Writing: Locating Parameters

Trick: How to locate string parameter (e.,g., "/bin/sh")

- Insert `call` right before `"/bin/sh"`
- Use `jmp` to jump to `call`
- `call` pushes %eip on stack
- After pop, address of `'/bin/sh'` in %eax

```
    jmp  short       L1
L2:
    pop              %eax
    ...
    ...
L1:
    call             L2
    .string          '/bin/sh'

```

Shellcode Writing: Example Linux/x86

```
 1 jmp    32                 // relative jump (to line 14)
 2 pop    %esi               // pointer to "/usr/bin/vim" now in %esi
 3 xor    %eax, %eax
 4 movb   $0x0, 0xc(%esi)    // prepare arguments for sys_execve
 5 mov    %esi, 0xd(%esi)
 6 movl   $0x0, 0x11(%esi)
 7 mov    %esi, %ebx
 8 lea    0xc(%esi), %ecx
 9 lea    0xd(%esi), %edx
10 movb   $0xb, %al          // call to sys_execve via int80
11 int    $0x80
12 movb   $0x1,%al           // call to sys_exit via int80
13 int    $0x80
14 call   -36                // relative call (line 2), pushes %eip
15 "/usr/bin/vim"
```

Stack Smashing Mitigations And Circumvention

Buffer Overflow: Some Dangerous C Standard Library Functions

- `strcpy` - copy buffers
- `memcpy`, `memmove` - copy buffers
- `strcat` - join 2 strings
- `sprintf`, `vsprintf` - print a string into another string
- `getpw` - reconstruct password-line entry
- `gets` - read a string from `stdin`
- `scanf` - read and convert a string from `stdin`
- `fscanf` - read and convert a string from a file pointer
- Pointer arithmetic

Buffer Overflow: More Dangerous C Library Functions

- Safer Alternatives:

  `strncpy`, `strncat`, `snprintf`, `vsnprintf`, `fgets`
- Wide Character Strings:

  `wcscpy`, `wmemcpy`, `wcscat`, `wcsncpy`, `fgetws`
- Conversion:

  `wcstombs`, `mbtowc`, `asctime_s`, `ctime_s`, `c16rtomb`, `c32rtomb`
- Non-ISO C:

  `read`, `bcopy`, `strlcpy`, `strlcat`,

Buffer Overflow Countermeasures: Developer and Tester

- Correct and secure programming paramount
- Correct input validation and length-verification
- Test for buffer overflow vulnerabilities
  - Static code analysis
  - Dynamic methods
  - Fuzz testing
  - Hybrid Methods
  - Code review
- Avoid dangerous functions (Use variants: `strncpy`, `strncat`, ...)
- Use type-safe programming languages

Buffer Overflow Countermeasures: Compiler and OS

- Non-executable stacks and heap
  - Data Execution Prevention (DEP)
  - W$\oplus$X: Write XOR Execute
- Randomized memory layout
  - Address Space Layout Randomization (ASLR)
- Compiled-in stack protection
  - Stack canary
  - Variable re-ordering

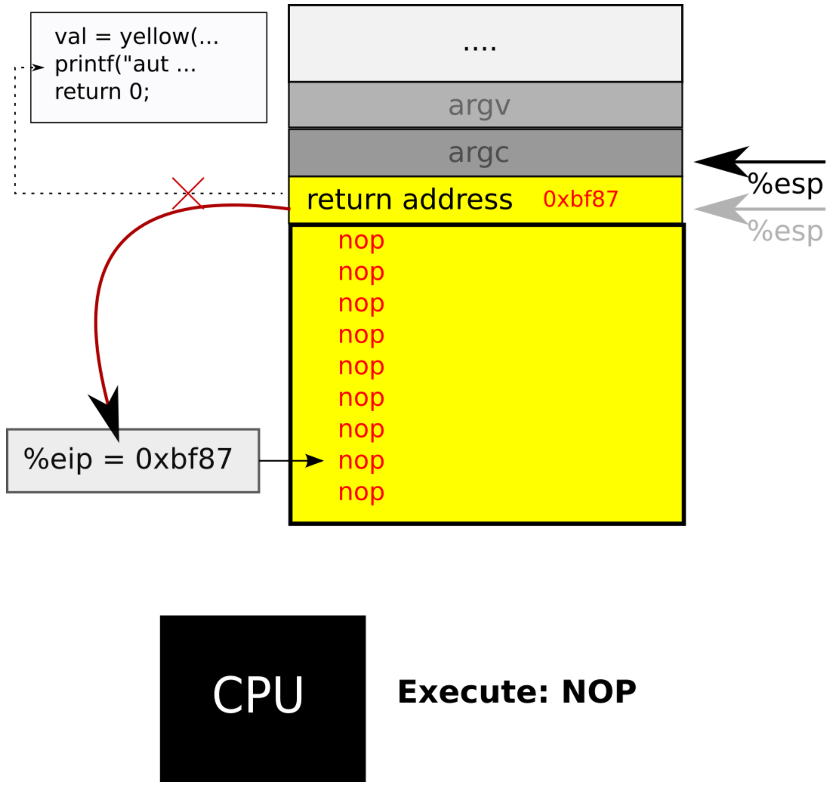
Buffer Overflow Countermeasures: Advanced

- Shadow Stack
- Pointer Integrity
- Control-flow Integrity
- Fine-Grained ALSR

Implement research prototypes as part of your 'Projektpraktikum' (12 ECTS) and/or Master Thesis.

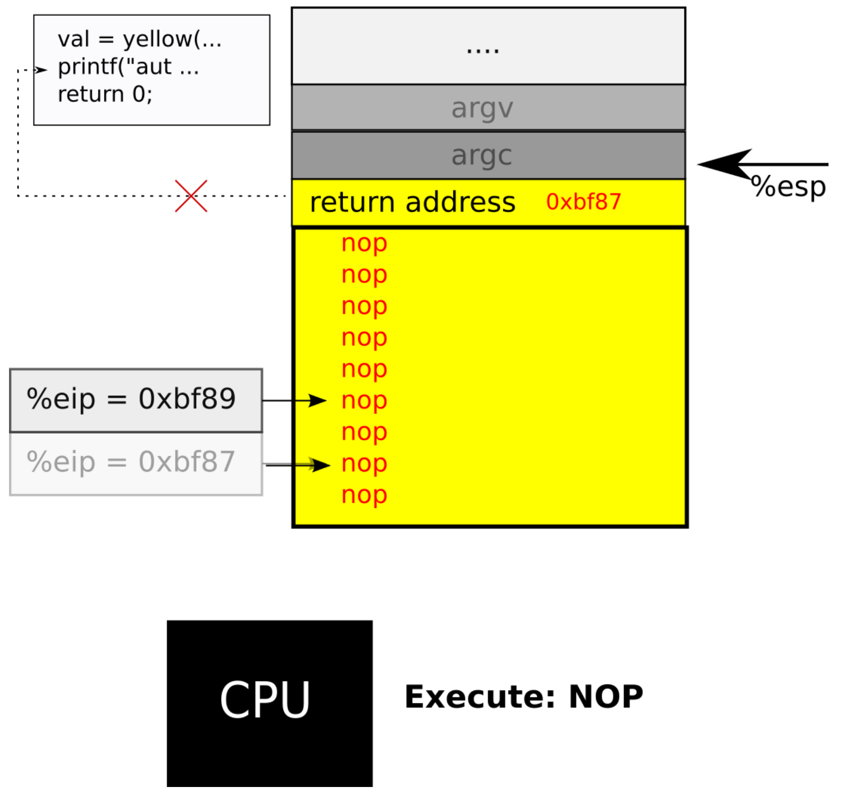
Stack Smashing Recap

1. Return address overwritten with address pointing inside `buf` on stack.
2. During function return (`ret` instruction), return address gets popped into `%eip` register
3. Instruction pointer (`%eip`) points into stack
4. Data on stack is interpreted as CPU instruction and executed



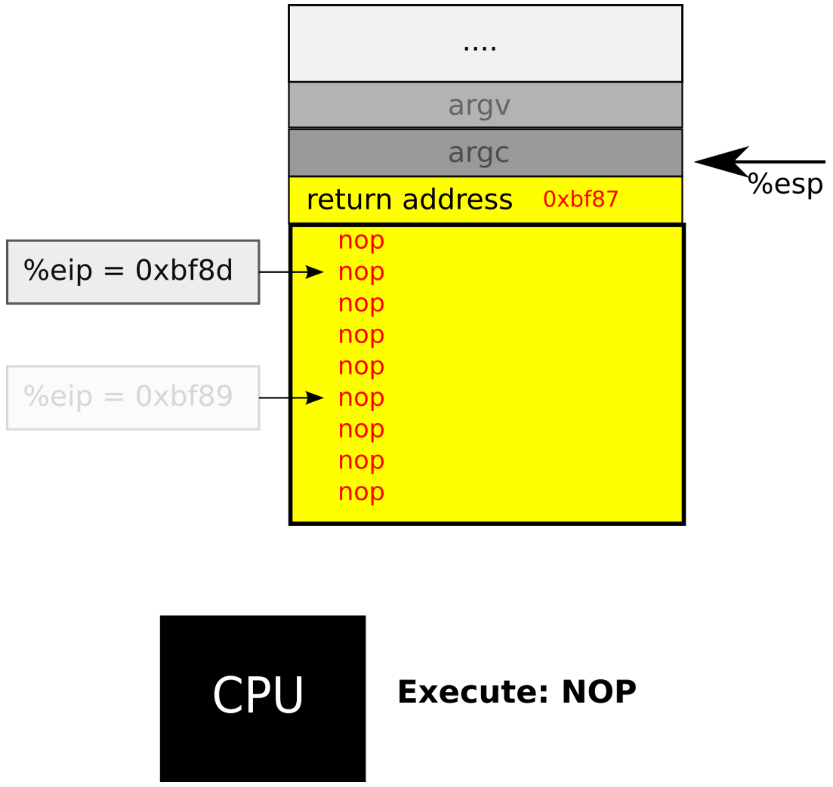
Stack Smashing Recap

- Non-control flow instructions increment the instruction pointer (`%eip`), so that it points to the next instruction
- Data at higher address is interpreted as instruction and executed



Stack Smashing Recap

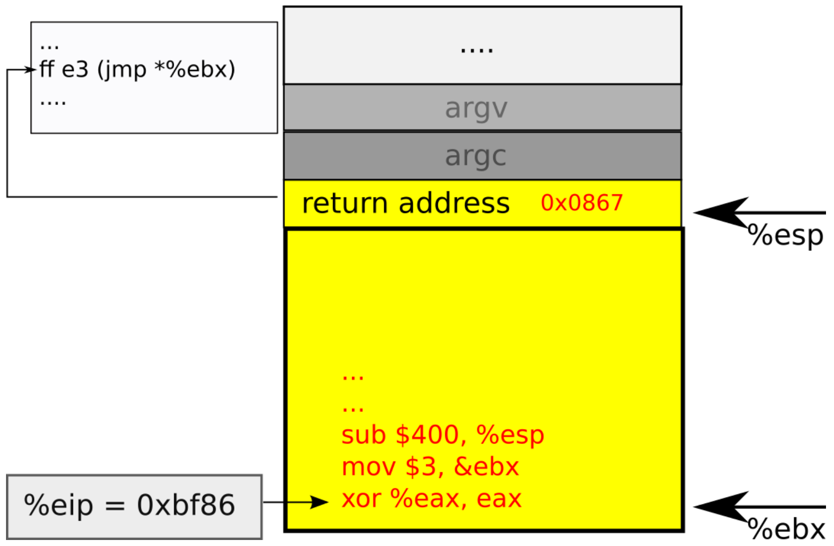
- Non-control flow instructions increment the instruction pointer (`%eip`), so that it points to the next instruction
- Data at higher address is interpreted as instruction and executed



Stack Smashing: jmp %ebx Trick

Assume on function return, any register (e.g., `ebx`), points to beginning of buffer

- Locate opcode for `jmp *%ebx` in process' memory
- Overwrite return address with location of this opcode

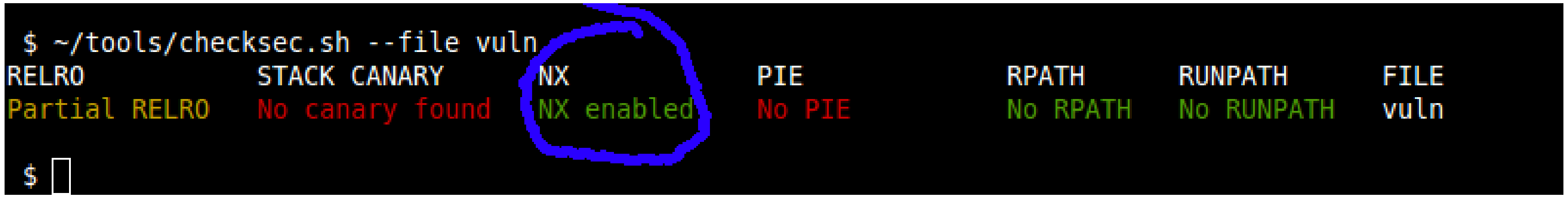


- Reliable jump into shellcode without NOP-Sled.
- Useful if buffer is too small for NOP-Sled

W⊕X Protection

Write XOR eXecute protection (as part of DEP in Windows)

- Memory region is either writeable or executable, but not both
- Prevent any user-injected code from being executed
- Hardware Support: NX Bit

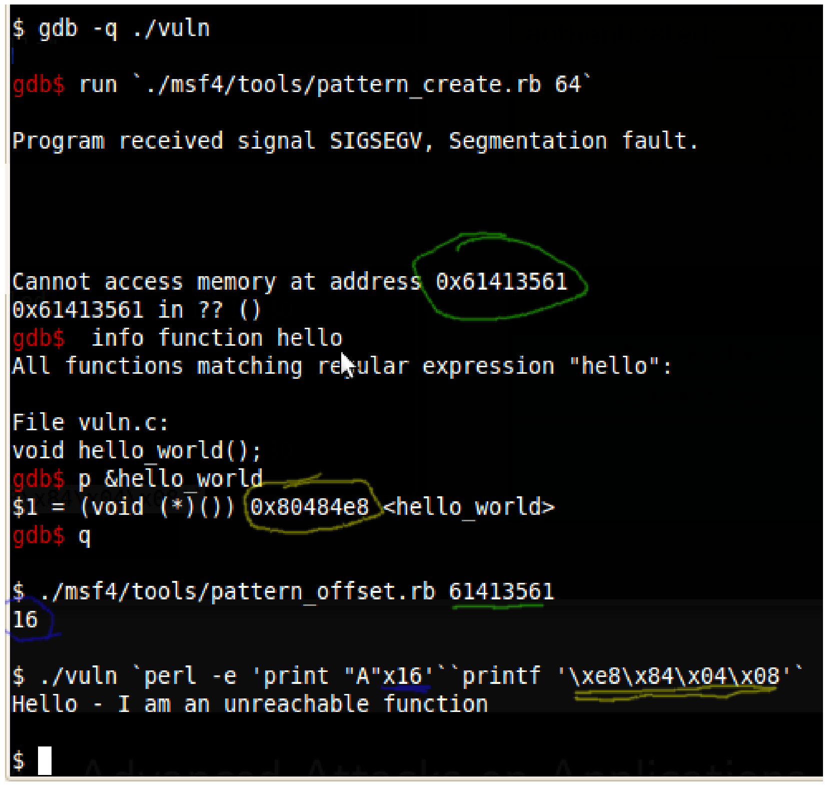


```
 $ ~/tools/checksec.sh --file vuln
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   vuln

 $ 
```

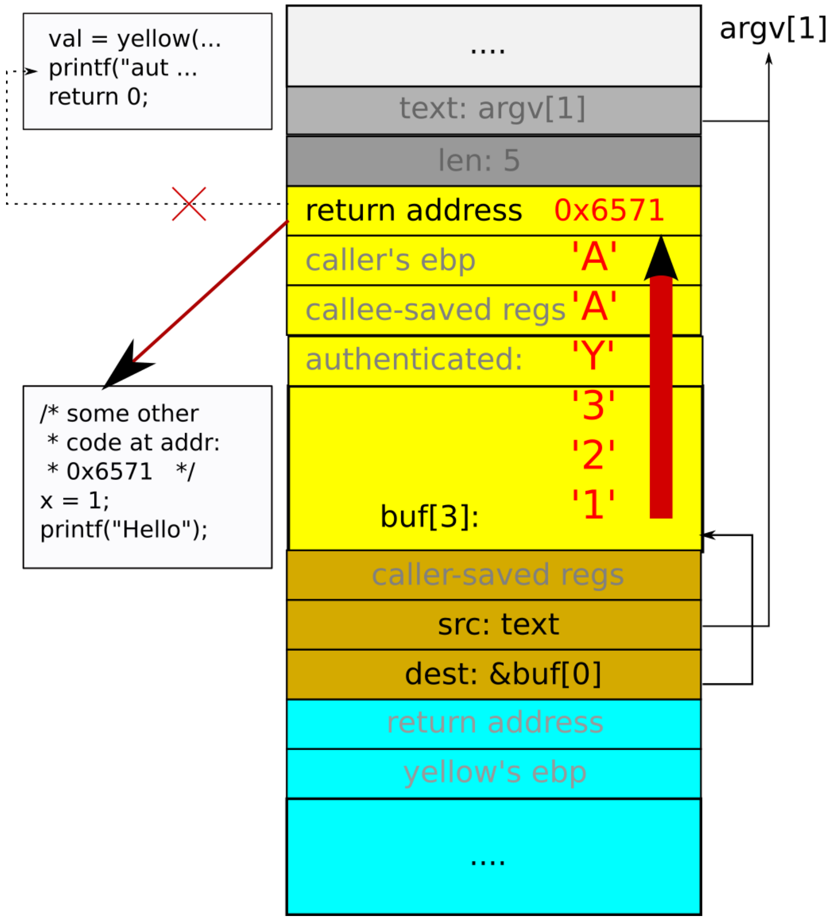
Circumvention: Return-into-text, Return-into-libc, ROP

W⊕X Circumvention: Return-into-text

Redirect control flow to a useful function in the .text (code-) section

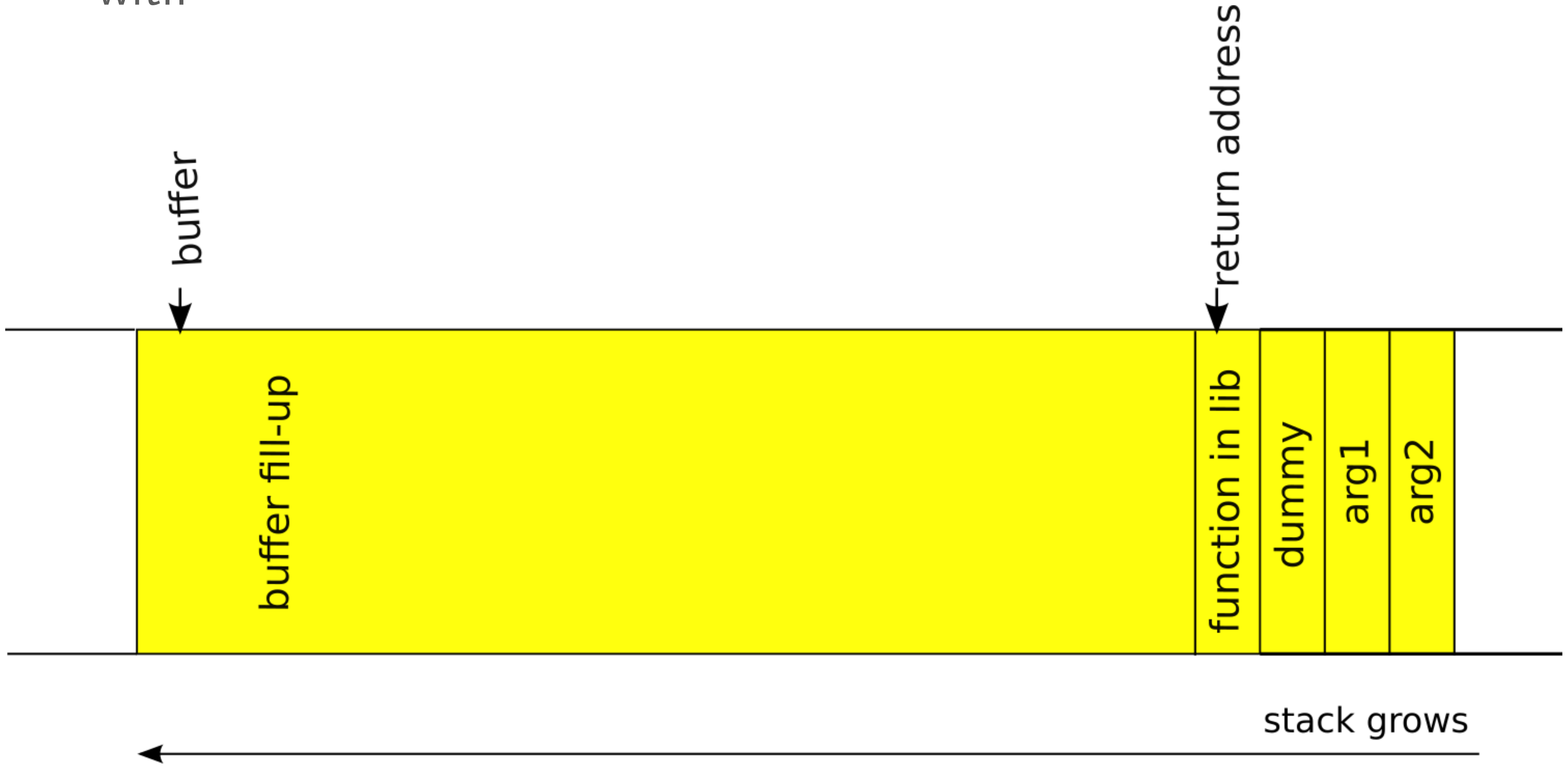


```
$ gdb -q ./vuln
gdb$ run `./msf4/tools/pattern_create.rb 64`

Program received signal SIGSEGV, Segmentation fault.



Cannot access memory at address 0x61413561
0x61413561 in ?? ()
gdb$  info function hello
All functions matching regular expression "hello":

File vuln.c:
void hello_world();
gdb$ p &hello_world
$1 = (void (*)()) 0x80484e8 <hello_world>
gdb$ q

$ ./msf4/tools/pattern_offset.rb 61413561
16

$ ./vuln `perl -e 'print "A"x16'``printf '\xe8\x84\x04\x08'`
Hello - I am an unreachable function

$ 
```



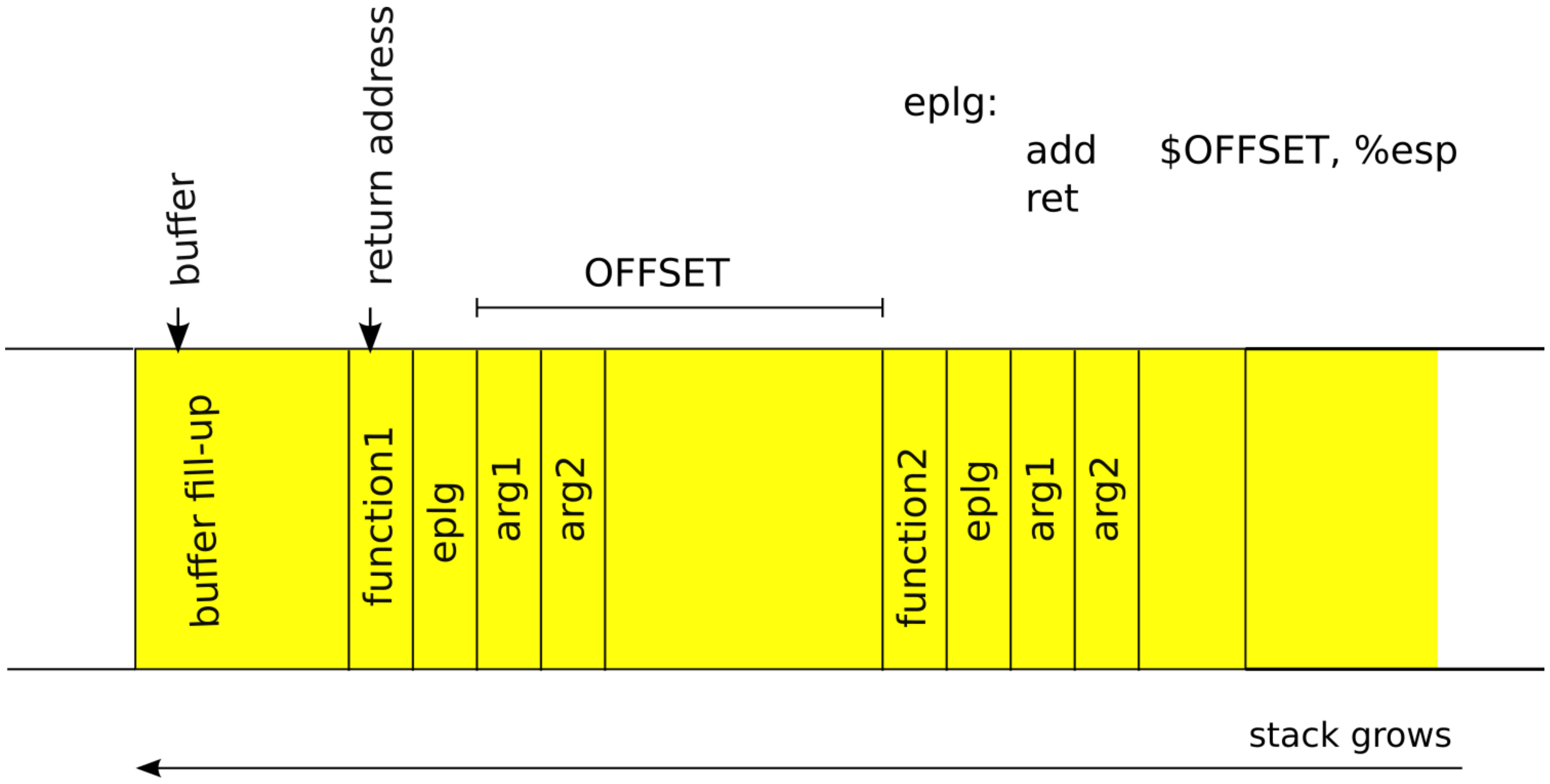
W⊕X Circumvention: Return-into-libc

- We can return into any function of any library the process is linked with



Return-into-libc: Function Chaining

Several function can be chained together.

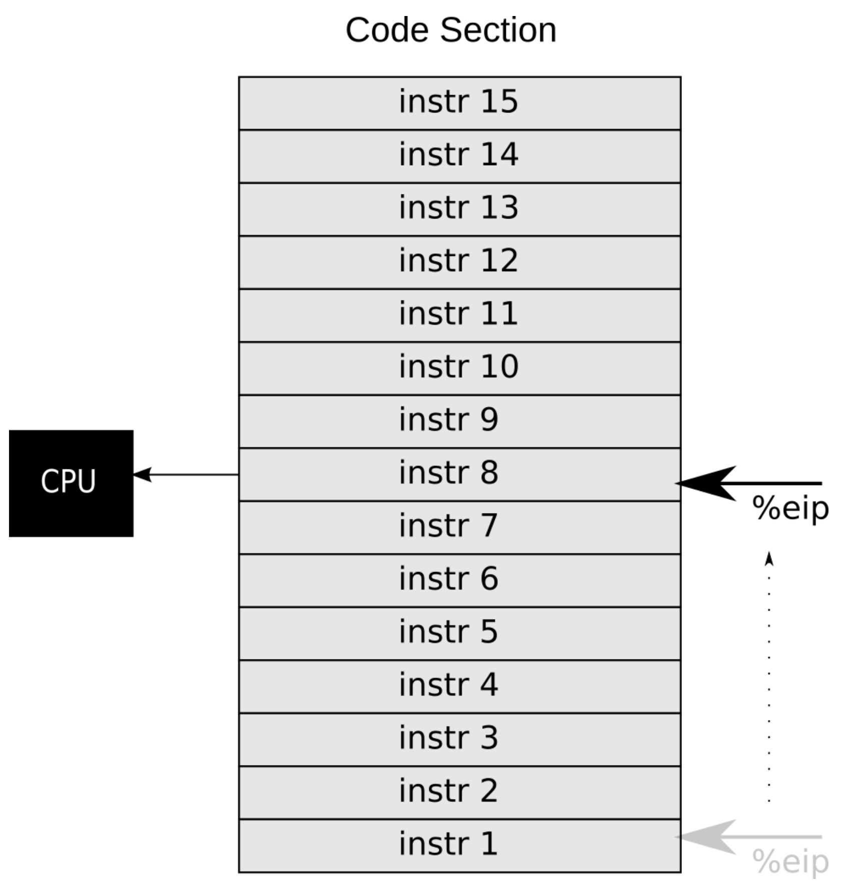


Return Oriented Programming (ROP)

- ROP takes Return-into-libc to the next level
- Return-into-libc not always possible:
  - Parameters passed via registers (e.g., x86_64 arch)
  - Library mapping effectively randomized
  - Library address contains 0x00 byte
- Return into sequence of instructions (“gadgets”) ending with `ret`
  - `xor %eax, %eax ; ret`
  - `inc %eax ; ret`
- Achieve arbitrary (Turing complete) computation with gadget-chaining

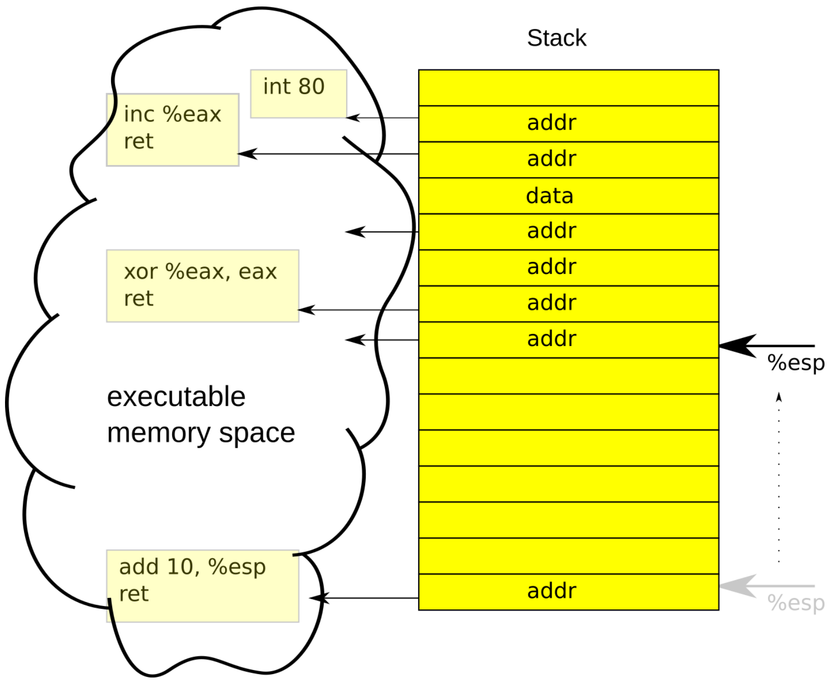
Ordinary Programming: Machine Level

- Instruction pointer (`%eip`) determines instruction to execute next
- Automatical increment of `%eip`
- Change control flow by changing value of `%eip`



Return Oriented Programming: Machine Level

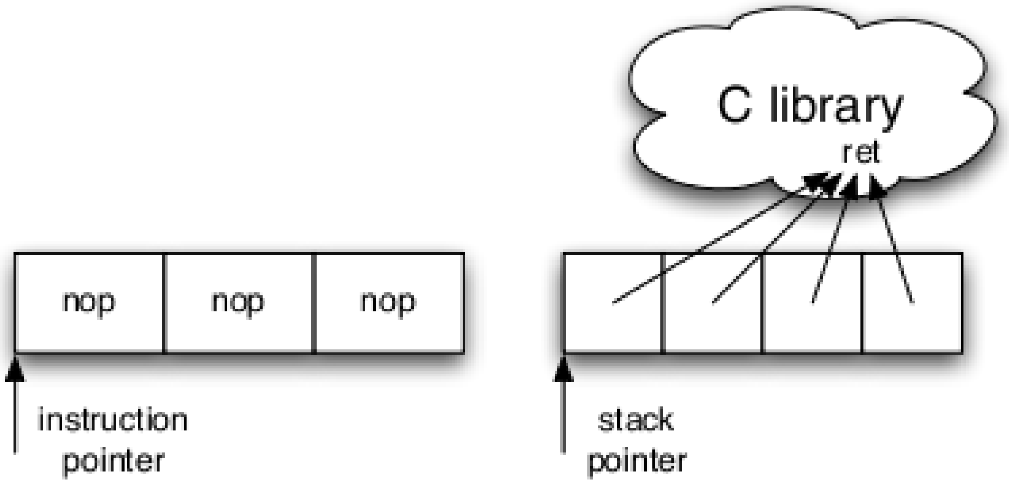
- Sequence of cpu instruction constitute logical instruction
- Stack pointer (%esp) determines next instruction to execute
- `ret` at end of gadgets increments %esp
- Control flow change by manipulating %esp



Return Oriented Programming: NOP Gadget

`ret`

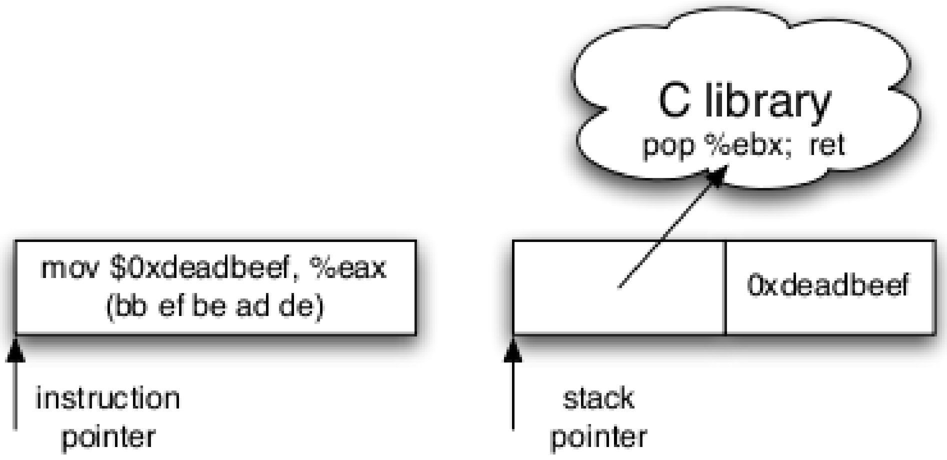
- A pointer to the opcode C3 (`ret`)



Ordinary and return-oriented nop sleds

Return Oriented Programming: Immediate Gadget

- A pointer to the code sequence `pop %ebx ; ret`
- `pop %ebx` will load the next dword into `%ebx`



Ordinary and return-oriented immediates

- `%esp` is incremented by both the `pop` and the `ret` instruction

esse

Return Oriented Programming

- Search for gadgets: upwards in code, starting from return instructions (Opcode `C3` on x86)
- Collect gadgets in TRIE data structure
- Automate ROP with compiler to produce Return Oriented Programs
- Python Tool to facilitate ROP exploitation: `ROPgadget`

Return Oriented Programming: Summary

- Turing complete
- Induce arbitrary behavior without injecting code.
- Defeat W⊕X, Code signing, Trusted Computing and Code Attestation, ...
- ROP vs return-into-libc: - ROP also works with different calling conventions (e.g., amd64: function arguments in registers)
- Works on different architectures: SPARC, ARM, ...
- However: base address of text, lib must be known.

ASLR

Idea: Defend against ROP, return-into-libc attacks

With each execution, randomize the

- load address of libraries and
- code-segment (text),
- the start address of the stack
- data segement, and
- the heap.

Defeating ASLR: 3 Strategies

- Process not fully randomized
- Determine address of library call by Brute Force
- Exploit an Information Leak

Beating ASLR: Process not fully randomized

- Executable must be compiled as Position Independent Executable (PIE)
  - Non-PIE binaries are protected only against trivial return-into-libc attacks
  - Otherwise: return-to-text, ROP
  - PIE: Performance overhead 5-10% on x86 (32 Bit)
- Any library at fixed address open possibility for ROP attacks.
  - Example: Linux Virtual Dynamic linked Shared Object (VDSO) (Interface to kernel space) was not randomized for a long time.

Beating ASLR: Brute Forcing ASLR

- Entropy on x86 too low to be effective: 10-16 Bit for library load address
- Bruteforce normally possible on x86 over network.
- Of course, process must still be alive after crash, or respawn.
- Processes that fork and afterwards call `execve` also exploitable: just one more Bit of entropy.

Beating ASLR: Information Leak

Use vulnerability to reveal memory content.

Examples:

- Windows: Modify BSTR length.
- Windows: Modify Array object length.

Overwrite length field of those objects to reveal memory content.

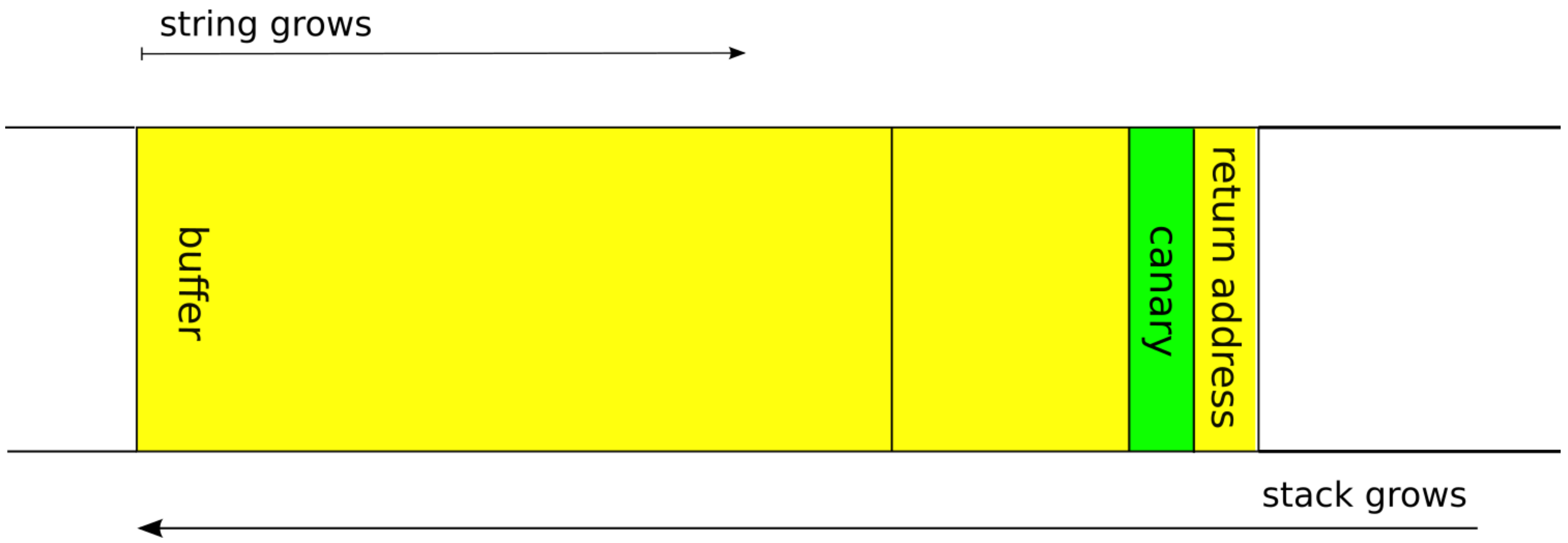
More Examples:

- Format String Vulnerability
- Out-of-Bound Read

esse

Stack Overflow Mitigation: Stack Canary

- During function prologue, a random canary value is placed after return address.
- Before function returns, canary value is checked and overflow detected.



Fixed Stack Canary vs Random Canary

- Fixed: `0x000a0dff`
  - Stops most string operations
  - Does little to prevent `memcpy` and direct pointer arithmetic corruptions
- Randomized
  - Different for each process execution
- Randomized XOR
  - Randomized and XOR control flow data

esse

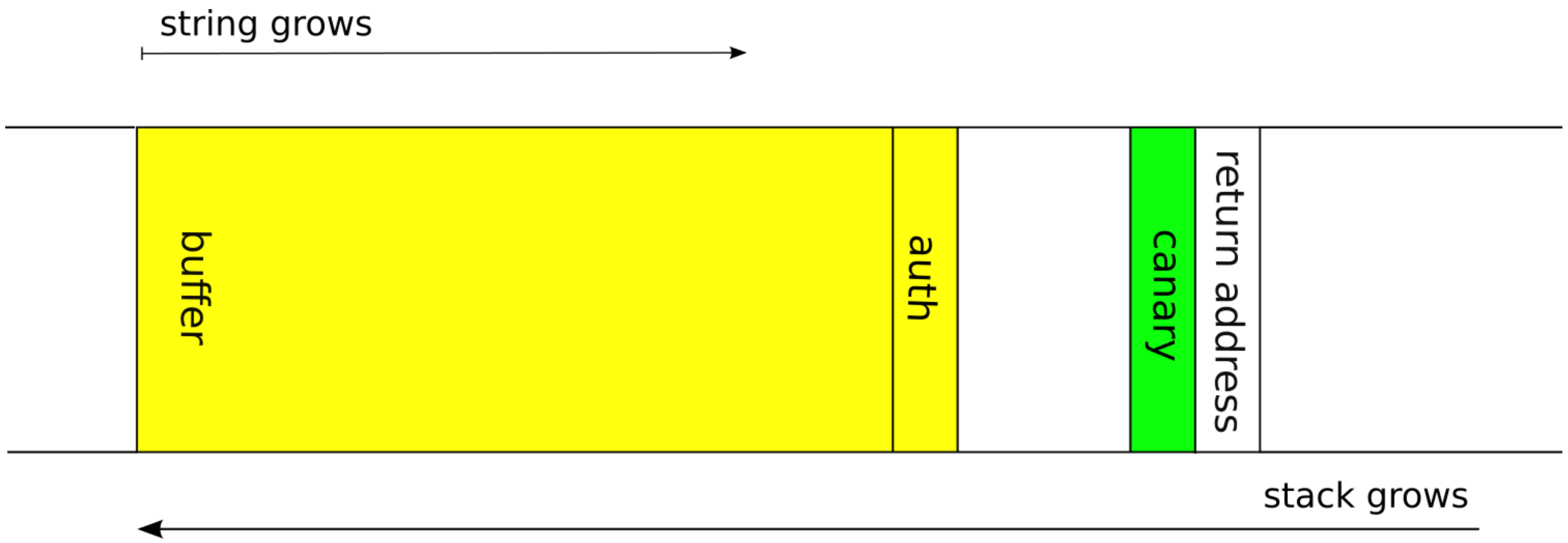
Stack Canary: Example Problem 1

What is the problem here?

```
int authenticate(char * username, char * password) {
    int auth = 0;
    char buffer[64];

    if (auth = verify(username, password) )
        sprintf(buffer, "succ auth: %s", username);
    else
        sprintf(buffer, "auth fail: %s", username);

        ...

    return auth;
}
```

Stack Canary: Local Variable Overwrite

- Return address need not be overwritten for successful attack
- Thus, canary value is never corrupted
- Attack succeeds without detection



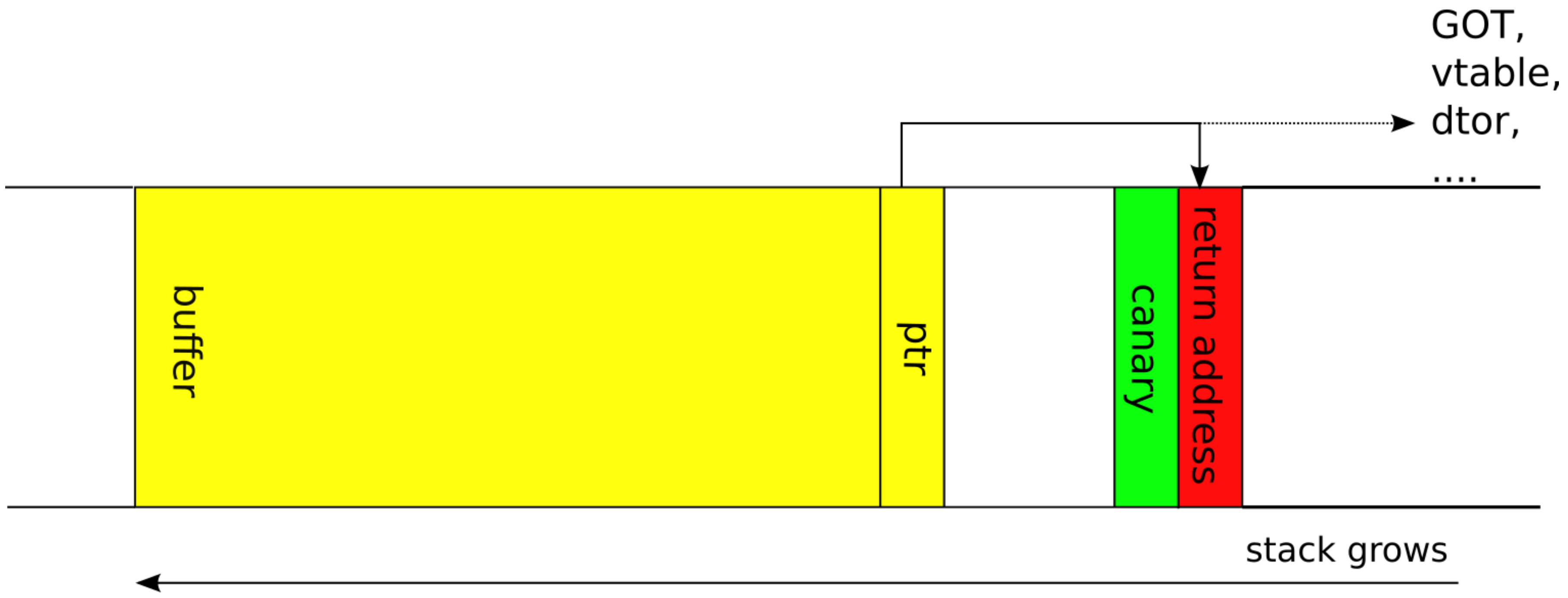
Stack Canary: Example Problem 2

What is the problem with this code?

```c
int f(char ** argv) {

    char *ptr;
    char buffer[64];

    ptr = buffer;
    memcpy (ptr, argv[1], 128);

    ...

    strncpy (ptr, argv[2], 16);

    ...
}
```

Stack Canary: Overwrite pointer

- Attacker can overwrite `ptr` with the `memcpy` call
- With the `strncpy` call, attacker can write to any memory location, without touching the canary



Stack Canary: Problem Example 3

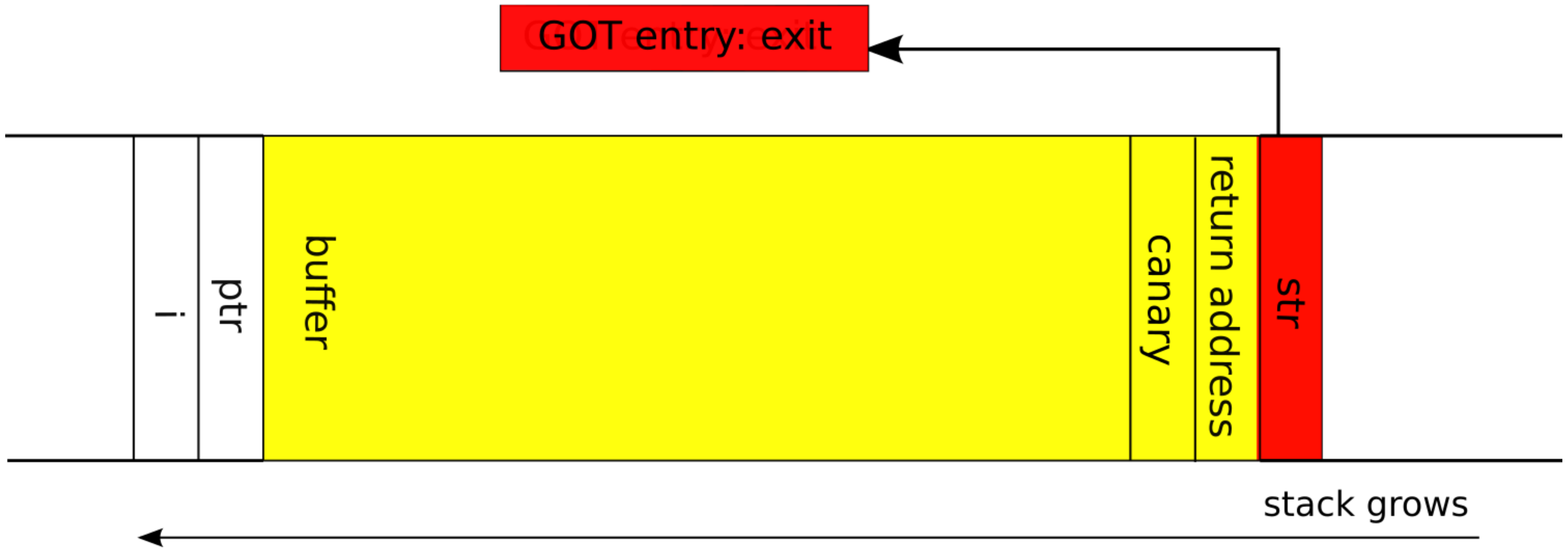
```
int g(char * str) {

    char buffer[64];
    char * ptr;
    int i;

    strcpy (buffer, str);
    ptr=buffer[0];

    for (i=0; i<64; i++) {
        *(ptr++)=toupper(*ptr);
    }
    strcpy (str, buffer);

    ...
    if (..) exit(1);
}
```

Stack Canary: Overwrite Argument

Attacker can overwrite the argument `str` with the first `strcpy`.

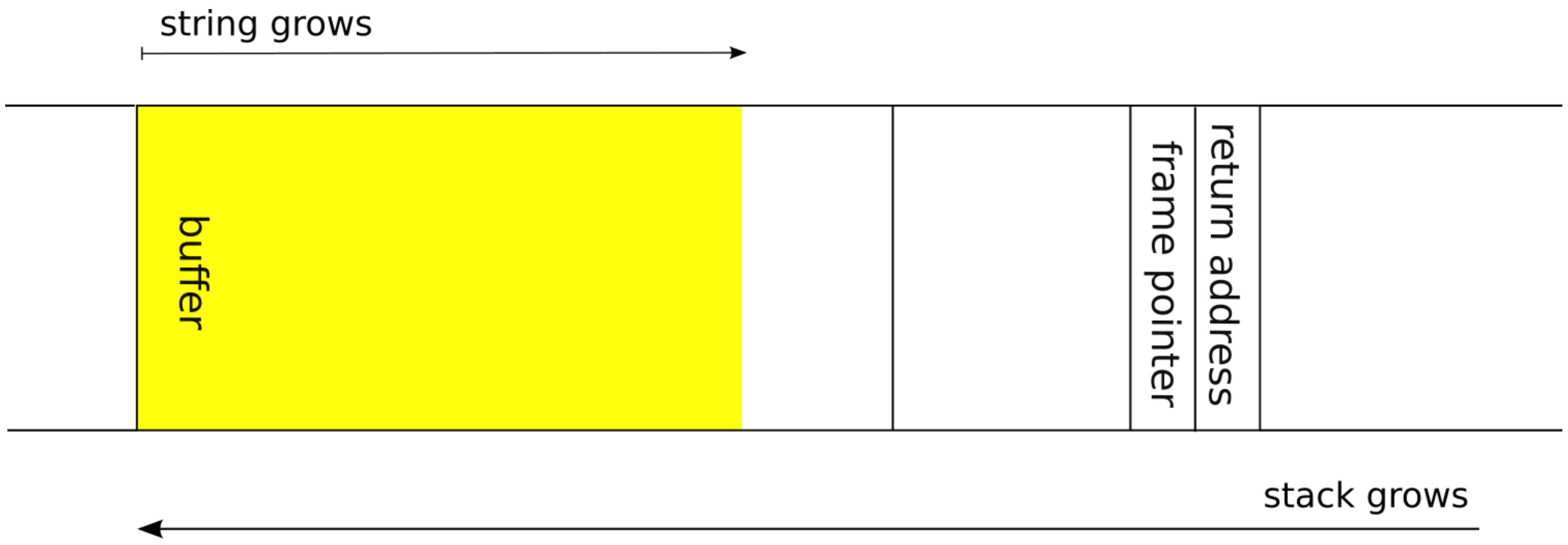
With the second `strcpy`, attacker can overwrite any address in memory.

Patching the `exit` call by overwriting the entry in the Global Offset Table (GOT) (Windows: Import Address Table (IAT)) allows gaining control of execution before the canary is checked.



Recap: Frame Pointer

Frame pointer of previous caller is stored just after return address.



Stack Protection: Frame Pointer

Frame pointer is used to access local variables and arguments.

- `%ebp+8` to address first argument
- `%ebp-offset` to address local variable

Overwriting the frame pointer (%ebp) thus “places” arguments and local variables to other memory region.
Thus, saved frame pointer needs protection too.

Stack Protection: Shadow Stack

Proposed Solution: Shadow Stack

- During function prologue, return address is saved on a “shadow stack”
- During function epilogue, return address is restored from shadow stack.

Can be trivially extended to protect the saved frame pointer.

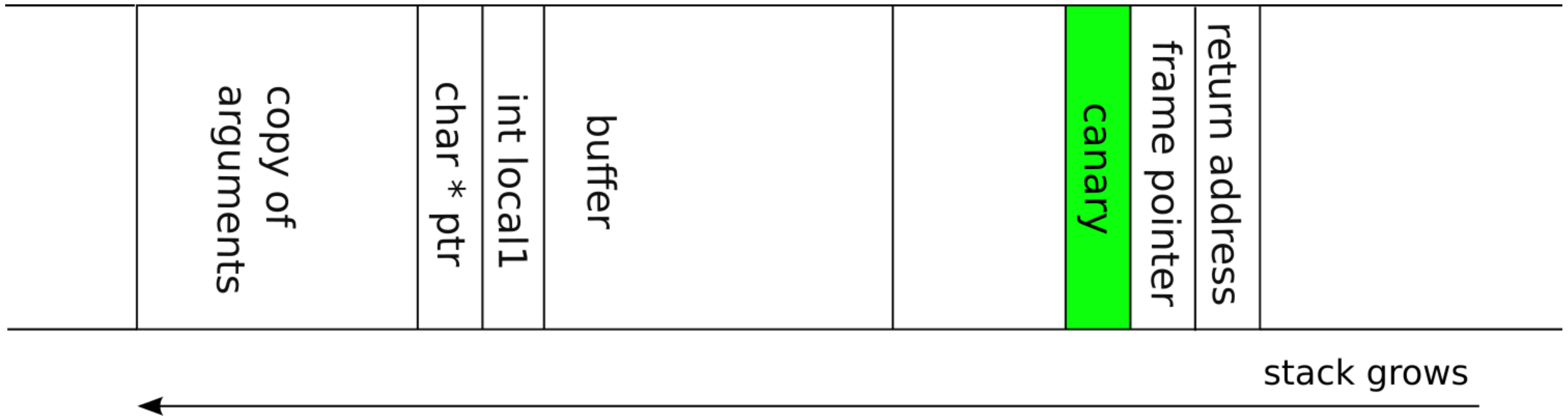
Stack Protection: Variable Re-ordering

Stack protection evolved to mitigate against these attacks.

- Buffers are placed after pointers
- Arguments are copied after local variables

However, stack buffer overruns remain still exploitable sometimes.

- Complexity; No simple solution



Bruteforcing Random Stack Canary: Preconditions

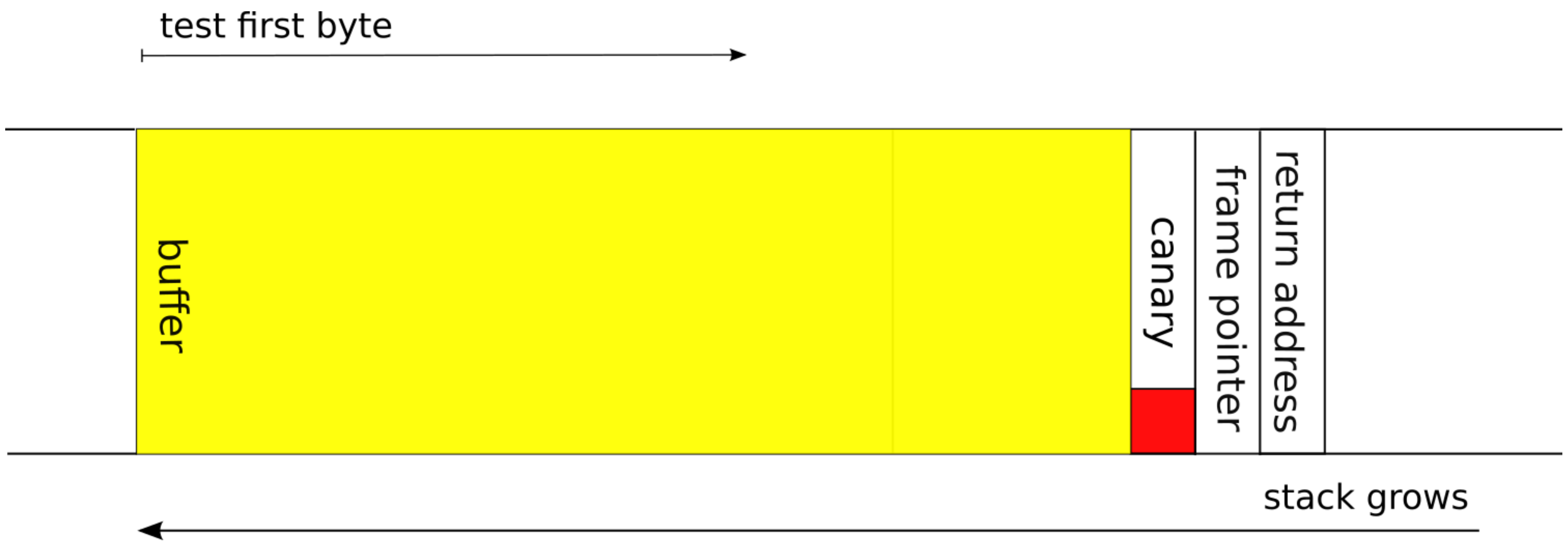
If

- a program forks (e.g., a network daemon to handle requests), memory space is copied, and
- the child process does not call `execve`, the randomized stack canary stays the same, and
- the attacker can determine whether his exploit crashed the child (via log-message, timing channel, etc.),

then the stack canary value can be easily determined.

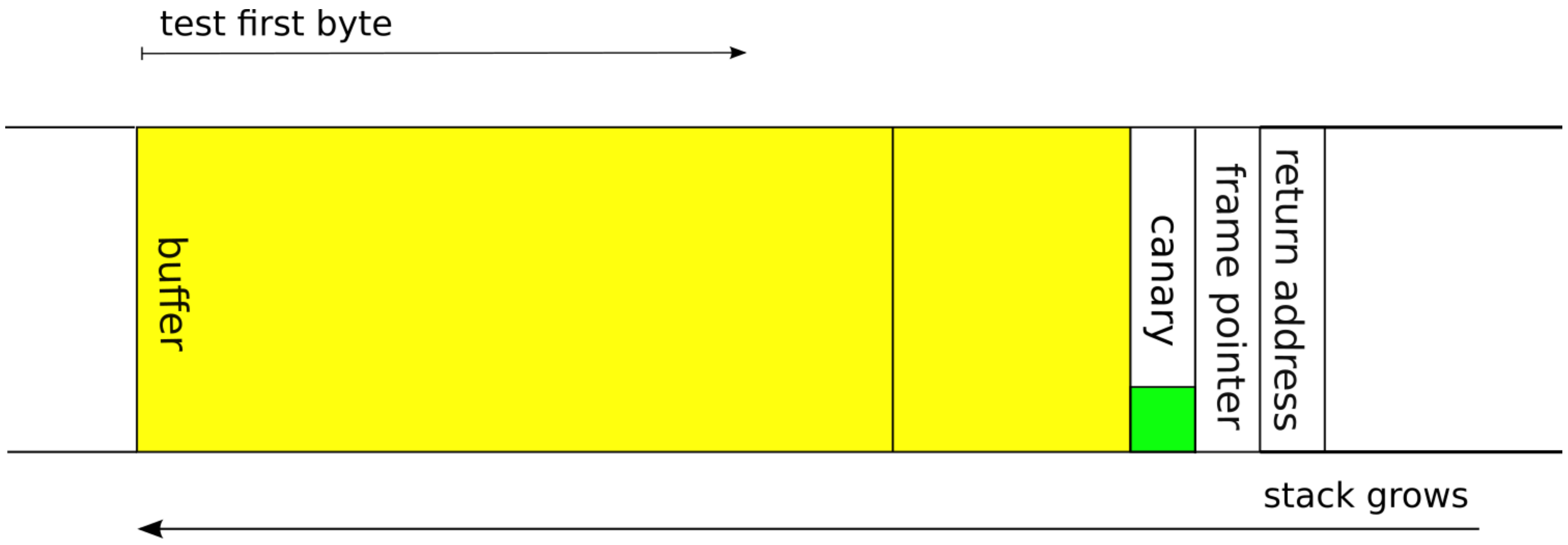
Bruteforcing Random Stack Canary

The buffer is overflowed until the first byte of the canary is corrupted. The attacker iterates over this last byte from $0 - 255$ by sending new exploits.



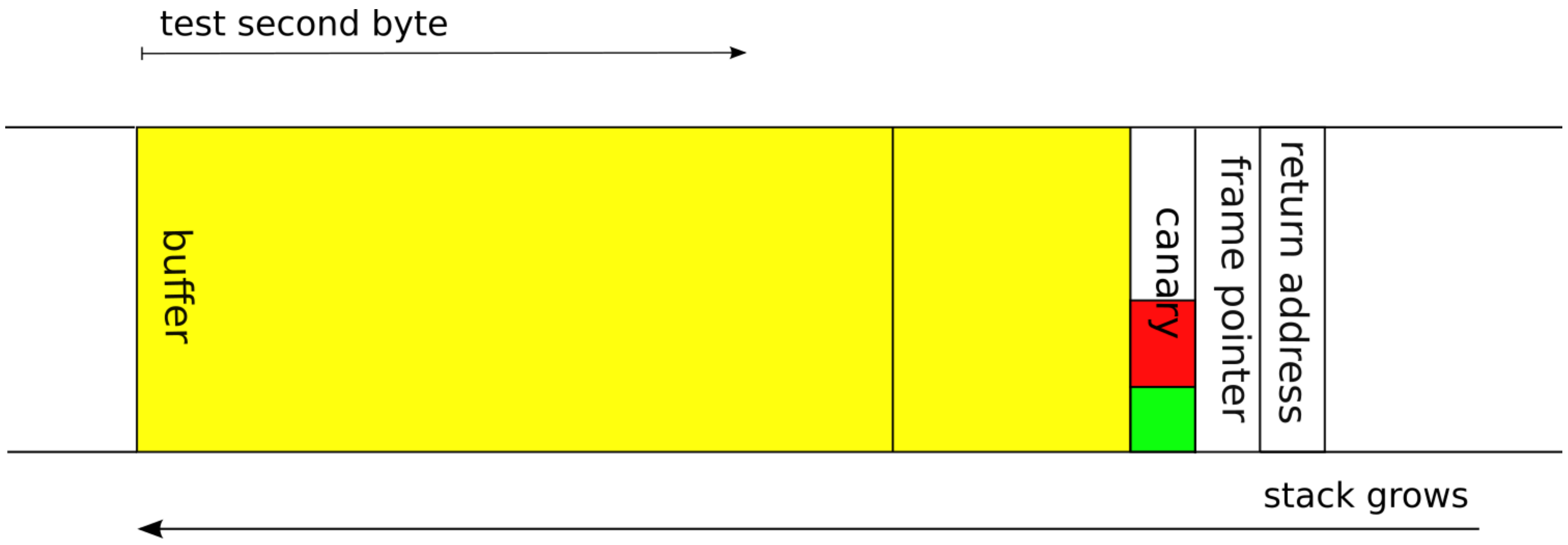
Bruteforcing Random Stack Canary

If the child process did not crash, he guessed the first byte correct.



Bruteforcing Random Stack Canary

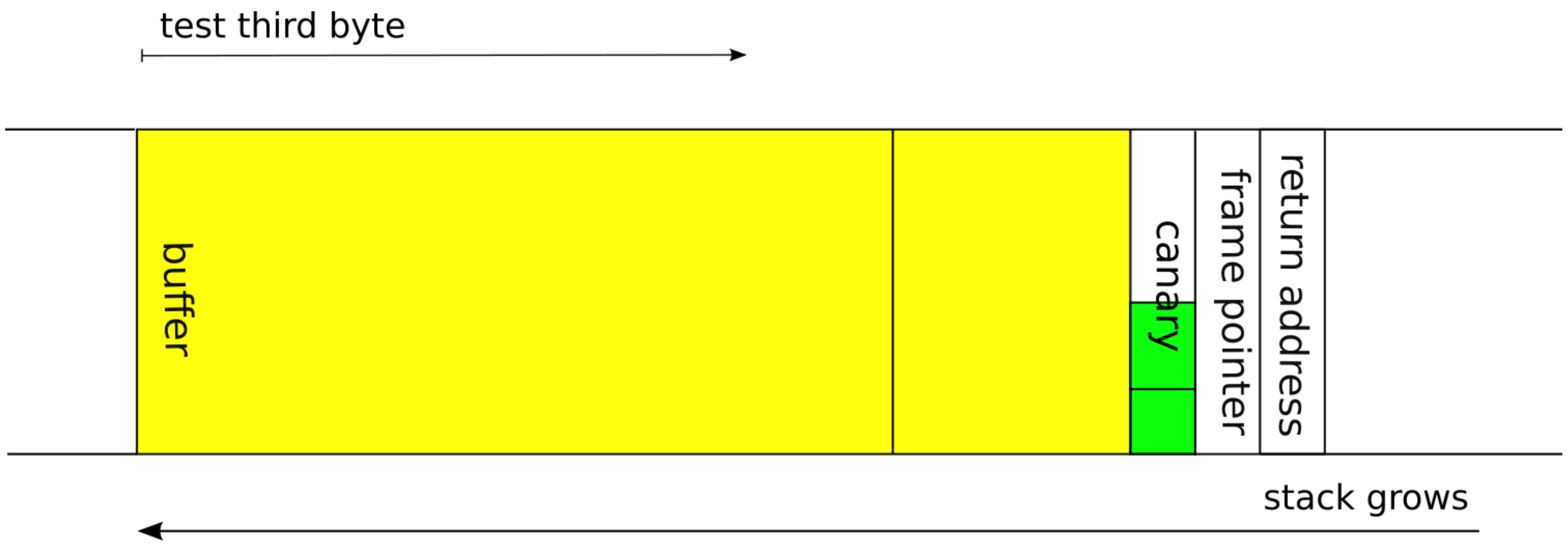
The attacker continues to probe the second byte.



esse

Bruteforcing Random Stack Canary

He eventually finds the second byte, and continues to probe the next byte, etc.
A 32 Bit canary can thus be found with max $4 \cdot 256$ tries ($4 \cdot 128$ expected value).



Overwrite Exception Handler

Windows stores pointers to exception handlers on the stack.

- Overwrite exception handler with new pointer
- If Exception is thrown before the function returns, attacker takes over program control before stack canary is checked.
- In many cases, the attacker is able to provoke an exception.

Mitigation: SafeSEH: All valid exceptions are registered in a function table.

Summary: Defeating Stack Protection

- Brute force canary

→ Works if process forks and child does not call `execve`

- Indirect write to arbitrary memory locations: RET, GOT / IAT, dtor, vtable

→ RELRO and BIND_NOW protects some of these locations in Linux

- Overwrite exception handler

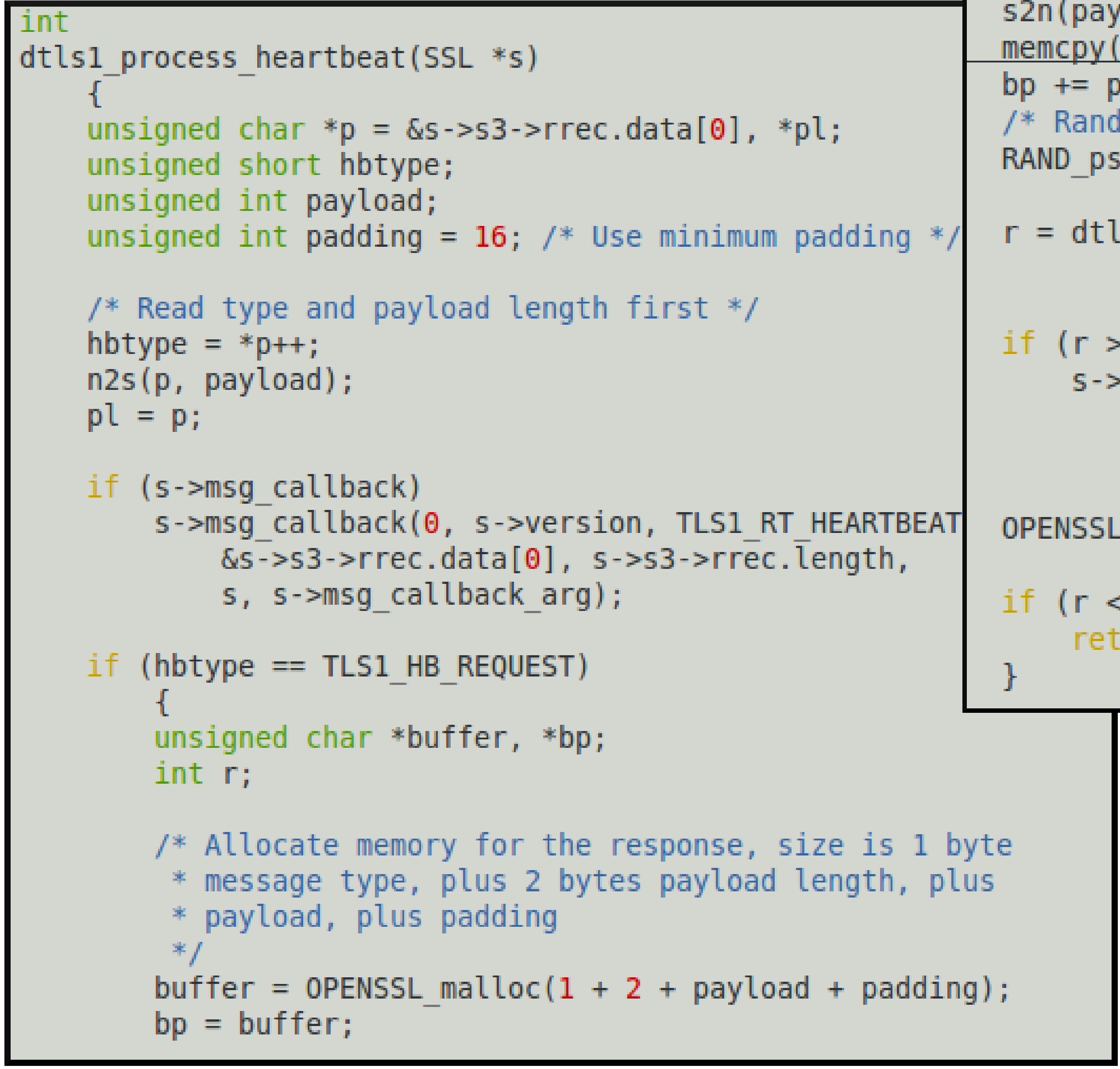
→ Easy if SafeSEH is not activated in Windows.

- For performance reason, usually not all functions are protected against stack overflows

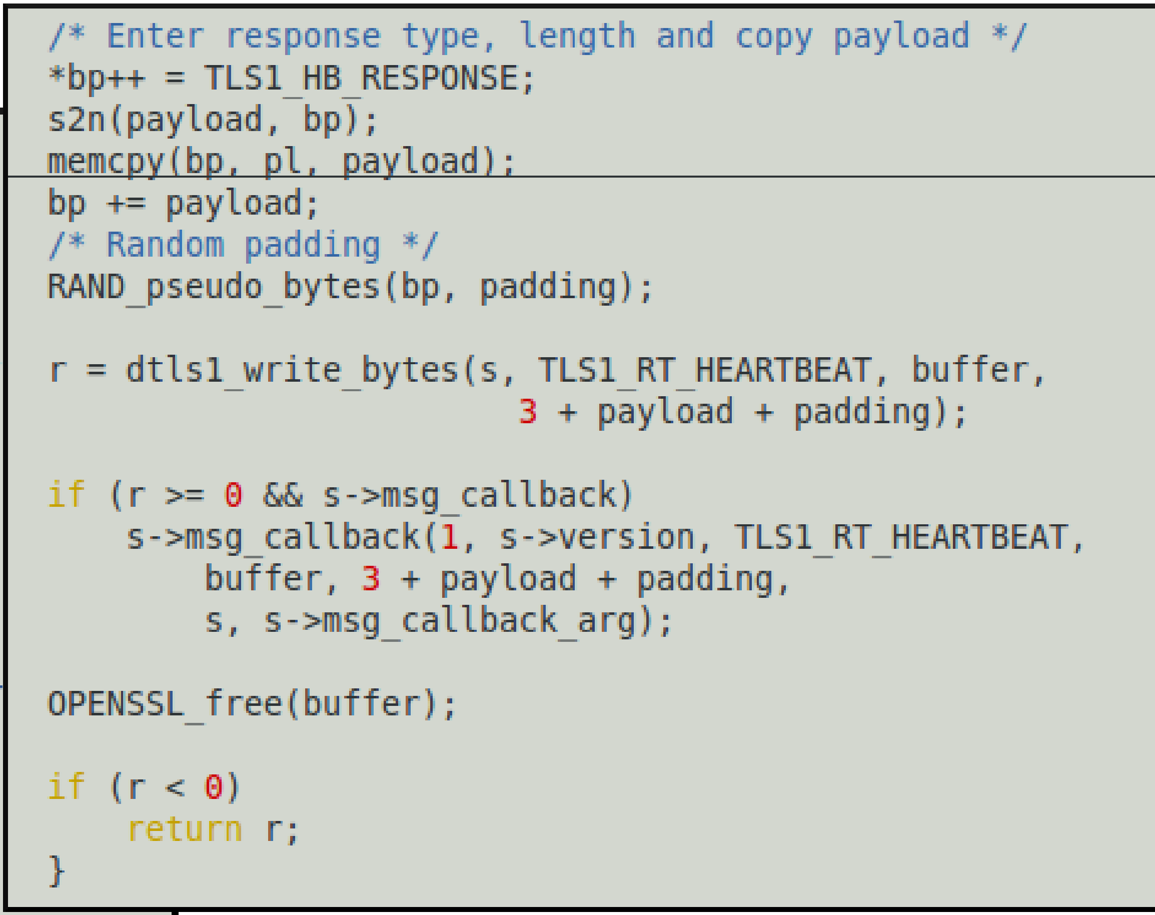
`-fstack-protector-all` vs. `-fstack-protector` vs. `-fstack-protector-strong` in gcc

More Classes of Vulnerabilites

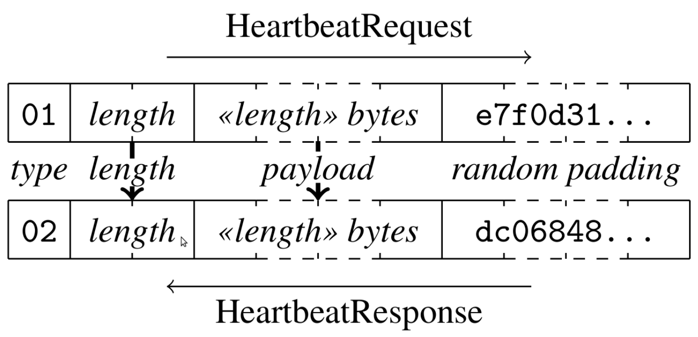
Heartbleed



```
int
dtls1_process_heartbeat(SSL *s)
    {
    unsigned char *p = &s->s3->rrec.data[0], *pl;
    unsigned short hbtype;
    unsigned int payload;
    unsigned int padding = 16; /* Use minimum padding */

    /* Read type and payload length first */
    hbtype = *p++;
    n2s(p, payload);
    pl = p;

    if (s->msg_callback)
        s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT
            &s->s3->rrec.data[0], s->s3->rrec.length,
            s, s->msg_callback_arg);

    if (hbtype == TLS1_HB_REQUEST)
        {
        unsigned char *buffer, *bp;
        int r;

        /* Allocate memory for the response, size is 1 byte
         * message type, plus 2 bytes payload length, plus
         * payload, plus padding
         */
        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
        bp = buffer;
```



```
/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
bp += payload;
/* Random padding */
RAND_pseudo_bytes(bp, padding);

r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer,
                      3 + payload + padding);

if (r >= 0 && s->msg_callback)
    s->msg_callback(1, s->version, TLS1_RT_HEARTBEAT,
        buffer, 3 + payload + padding,
        s, s->msg_callback_arg);

OPENSSL_free(buffer);

if (r < 0)
    return r;
}
```



Out-of-bounds Read

- Read beyond the memory of an allocated buffer
- Cause: Lack of correct bounds checking
- Information disclosure vulnerability, but can be disastrous

Heartbleed Bug (disclosed April 2014)

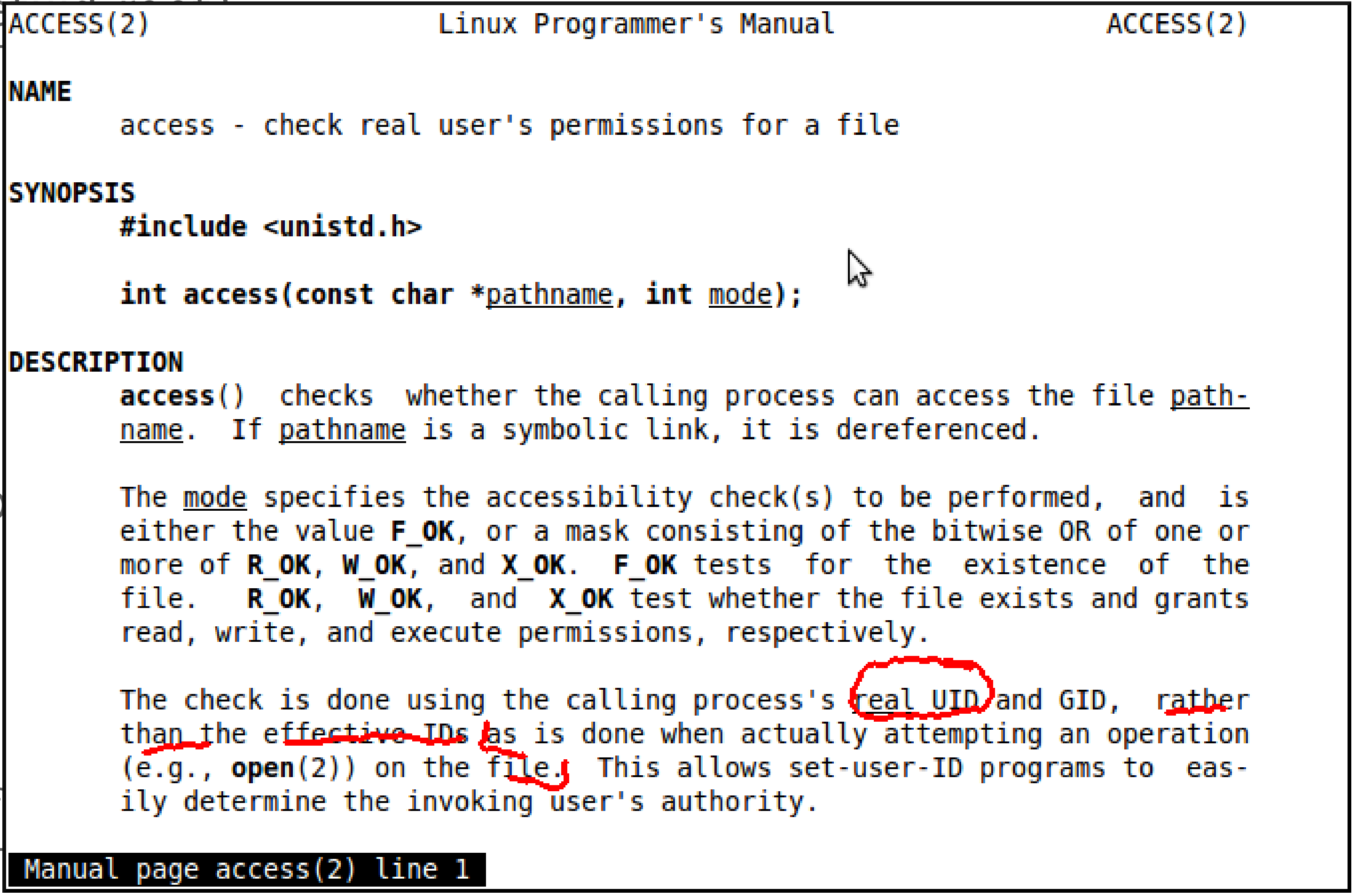
- Programming error in openssl library
- Length field in heartbeat packet attacker controlled
- No comparision with actual received record size
- Read up to 64Kb memory adjacent to `s->s3->rrec.data`
- Extract private keys, passwords, etc. from memory
- Heartbleed affected an estimated 24-55% of HTTPS server

Race Condition: access/open TOCTTOU

Victim (installed se

```
if (access("file",
{
    exit(1);
}



fd = open("file", O



write(fd, buffer,
        sizeof(buff
```

Unix access syscall



```
ACCESS(2)                  Linux Programmer's Manual                 ACCESS(2)

NAME
       access - check real user's permissions for a file

SYNOPSIS
       #include <unistd.h>

       int access(const char *pathname, int mode);

DESCRIPTION
       access()  checks  whether the calling process can access the file path-
       name.  If pathname is a symbolic link, it is dereferenced.

       The mode specifies the accessibility check(s) to be performed,  and  is
       either the value F_OK, or a mask consisting of the bitwise OR of one or
       more of R_OK, W_OK, and X_OK.  F_OK tests  for  the  existence  of  the
       file.   R_OK,  W_OK,  and  X_OK test whether the file exists and grants
       read, write, and execute permissions, respectively.

       The check is done using the calling process's real UID and GID,  rather
       than the effective IDs as is done when actually attempting an operation
       (e.g., open(2)) on the file.  This allows set-user-ID programs to  eas-
       ily determine the invoking user's authority.

 Manual page access(2) line 1
```

esse

Race Condition: access/open TOCTTOU

Victim (installed setuid-root)

```
if (access("file", W_OK) != 0)
{
    exit(1);
}



fd = open("file", O_WRONLY);

// Actually writing over
//  /etc/shadow
write(fd, buffer,
           sizeof(buffer));
```

Attacker

```


...
// After the access check
symlink("/etc/shadow", "file");
// Before the open, "file"
// points to /etc/shadow
...
```

Race Condition: access/open TOCTTOU

Victim (installed setuid-root)

```
if (access("file", W_OK) != 0)
{
    exit(1);
}



fd = open("file", O_WRONLY);

// Actually writing over
//  /etc/shadow
write(fd, buffer,
          sizeof(buffer));
```

Attack (general outline)

```
// Let the victim run
if (fork() == 0)
    system("victim");
usleep(1); // Yield CPU
//switch target
unlink("file")
symlink("/etc/shadow", "file");
```

Race Condition: TOCTTOU

Time Of Check Time Of Use races

- Concurrent processes: `setuid` program, privileged server
- Access to shared resources: filesystem, sockets, database
- Often difficult to spot and reproduce

Exploiting TOCTTOU races: scheduler needs to switch at the right instruction to attacker's process

- Bruteforce
- Filesystem maze
- Algorithmic complexity attacks

Race Condition: TOCTTOU Mitigation

Countermeasures examples:

- Kernel run-time detection and prevention (state management problem)
- Security test: data and control flow analysis tools
- Transactional file system
- Use not-portable, secure functions
- `fork/setuid/open` + IPC
- Atomic operations, concurrency control

Race Condition: CVE-1999-0861

Race condition in SSL / MS Internet Information Services (IIS)

- Sending an encrypted message
- Correct sequence
  1. load plain text into buffer
  2. encrypt buffer
  3. send data from buffer
- Error at high load
  1. load plain text into buffer
  2. send data from buffer
  3. encrypt buffer

Race Condition: Smartcards PIN Bruteforce

Smartcards need to protect against PIN bruteforcing:

1. Counter initialized to 3
2. If wrong PIN, then decrement counter
3. If counter 0, lock card
4. If PIN correct, reset counter

Secure?

Brute force attack, racing:

1. Enter PIN
2. Check if PIN is correct (e.g., using a side channel)
3. Before counter on card is decreased, pull power plug

Countermeasure:

- Decrease counter before checking PIN

esse

Format String Functions

```
printf("error in line %i: %s", linenr, errorstring);
```

- Use format control strings to generate output strings
- Functions: `printf` family (`fprintf`, `sprintf`, ...)
- Different format control characters, see `man sprintf`
  - integer: %d, %i, %o, %u, %x
  - float: %e, %f, %g, %a
  - character: %c
  - string: %s

Format String Vulnerability

- User data is directly passed to `printf`
- Attacker can provide format string
- Correct implementation is `printf("%s", argv[1]);`

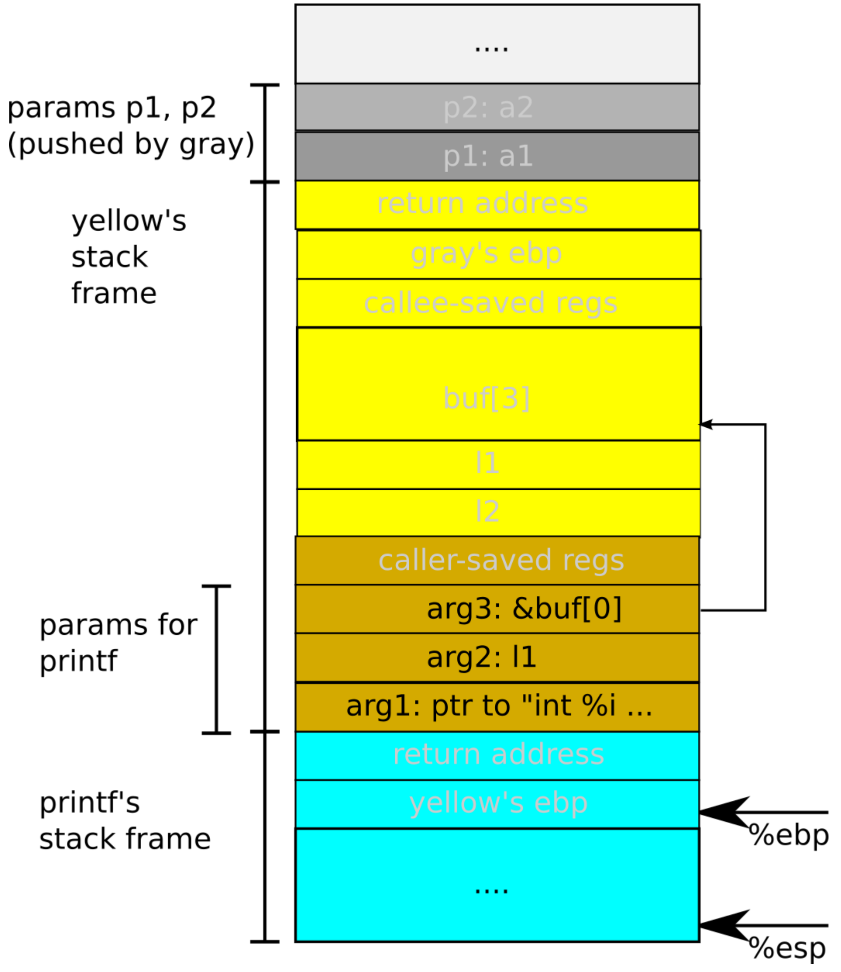
```c
#include <stdio.h>
#include <string.h>

int
main(int argc, char* argv[])
{
    printf(argv[1]);
}
```

- Easy to test for and to find automatically in many cases (compile-time checking)
- Dynamic format string generation, cross-application format string dangerous

esse

Recap: Stack Layout, Function Parameters

```
void gray()
{
    ...
    yellow(a1, a2);
    ...
}

int yellow(int p1, int p2)
{
    char buf[3] ;
    int l1, l2;
    ...
    l2 = printf("int: %i, str: %s", l1, buf);
    return l2;
}
```



Format String Vulnerability: Read Stack Content

- Format function assumes all parameters are correctly pushed to the stack
- Attacker can read the whole stack content

```
$ ./format AAAA.%x.%x.%x
AAAA.8049ff4.bffff8b8.8048459

$ ./format "AAAA.%3$x"
AAAA.8048459

$ ./format `perl -e 'print "%08x." x1000'`
...41007461.41414141.25414141.2e783830...
```

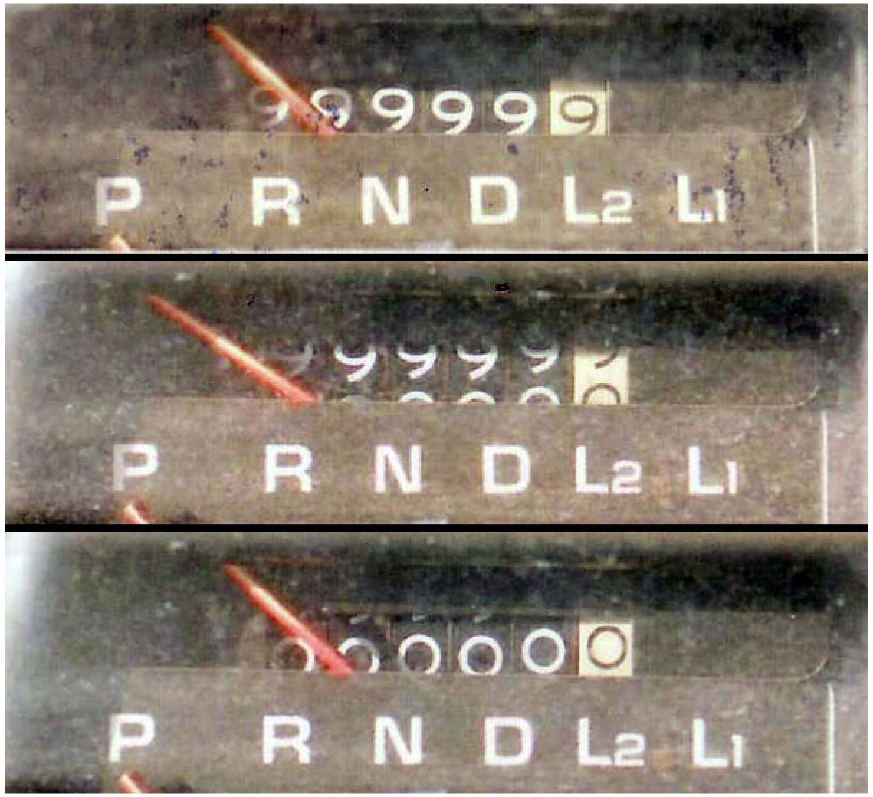
Parameter %n allows write: Control flow hijacking possible

Format String Vulnerability

- Read (arbitrary) memory locations
  - Confidential informations in memory
  - Confidential keys, passwords, . . .
- Write to any memory location
  - Overwrite addresses (return address, . . .)
  - Control flow hijacking
  - Arbitrary code execution

Integer Overflow

- n-bit (register!) arithmetic $\neq$ (∞-bit) arithmetic
- Many languages: C, C++, Java, C#, Go
- Sometimes intentional
- Sometimes very difficult to catch



Integer Overflow: Example Intentional Wrap

Linear Congruential PRNG

```c
#define IA 1103515245u
#define IC 12345u
#define IM 2147483648u

static unsigned int c_rand = 0;

/* Creates a random integer [0... imax ] (inclusive) */
int my_irand ( int imax ) {
     int ival ;
     /* c_rand = ( c_rand * IA + IC ) % IM ; */
     c_rand = c_rand * IA + IC ; // Use overflow to wrap
     ival = c_rand & ( IM - 1) ; /* Modulus */
     ival = ( int ) (( float ) ival * ( float ) ( imax + 0.999)
                        / ( float ) IM ) ;
return ival ;
}
```

Integer Overflow: Malformed Checks

```c
struct DS {
    ...
    int num;
    int values[];
}
...
// this check is malformed
if (num > INT_MAX / sizeof(int)
                 - sizeof(DS))
    goto fail;
// heap overflow possible
... = malloc(sizeof(DS) +
           num * sizeof(int));
```

- Magic values hard to maintain: datastructure might change!
- Often: Incorrect check
- Check to avoid overflow of form $a + x * b > MAX$:

$$x > (MAX - a)/b$$

Integer Vulnerability

Netscape vulnerability

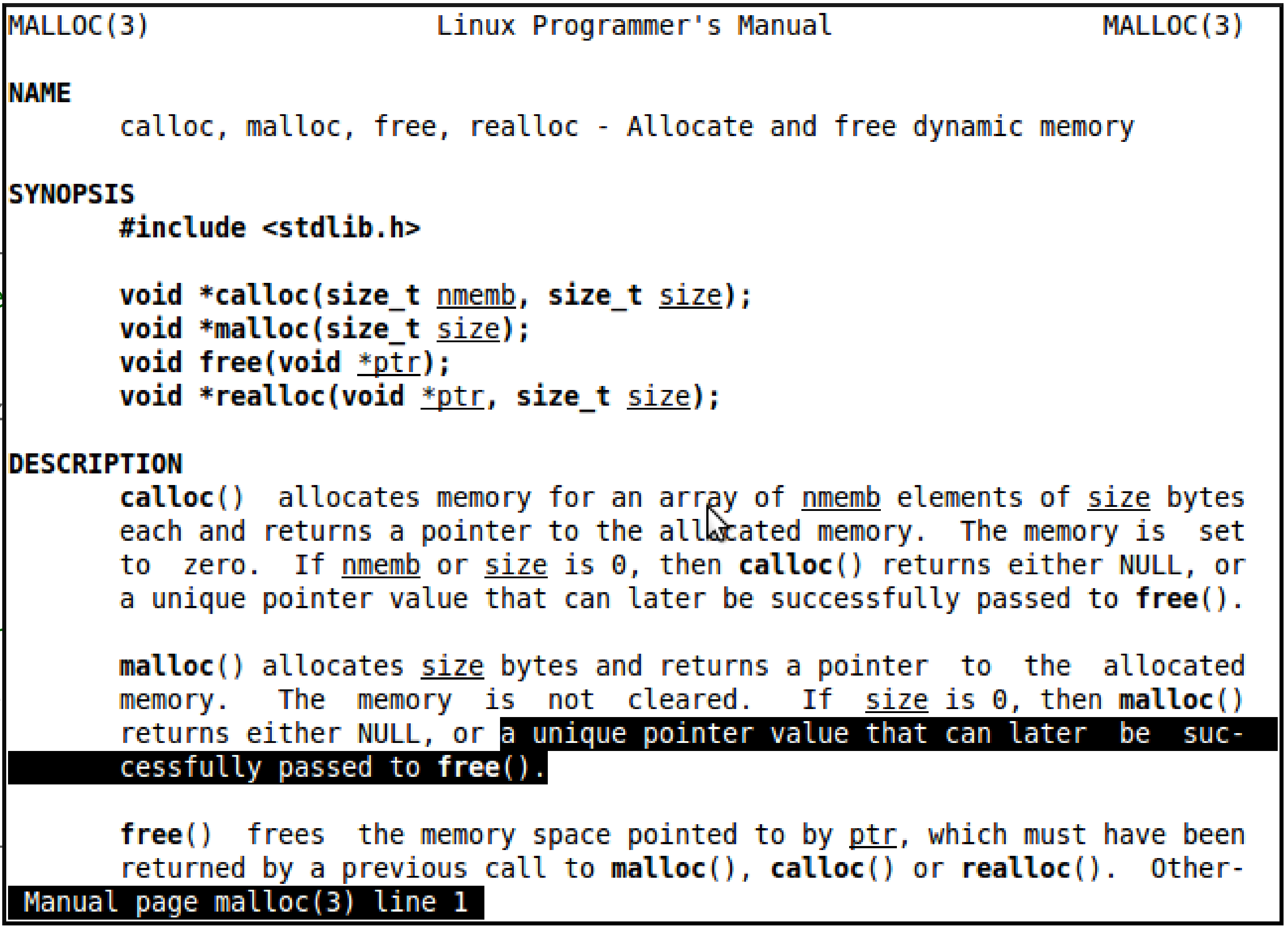
```
void getComm(unsigned int len, char *src)
{
    unsigned int size;

    size = len - 2;

    char *comm = (char *)malloc(size + 1);
    memcpy(comm, src, size);
    return;
}
```

What can go wrong here?

Integer Vulnerability: Underflow

Netscape

```
void getComm(unsigne
{
    unsigned int siz

    size = len - 2;

    char *comm = (ch
    memcpy(comm, src
    return;
}
```



```
MALLOC(3)                  Linux Programmer's Manual                 MALLOC(3)

NAME
       calloc, malloc, free, realloc - Allocate and free dynamic memory

SYNOPSIS
       #include <stdlib.h>

       void *calloc(size_t nmemb, size_t size);
       void *malloc(size_t size);
       void free(void *ptr);
       void *realloc(void *ptr, size_t size);

DESCRIPTION
       calloc()  allocates memory for an array of nmemb elements of size bytes
       each and returns a pointer to the allocated memory.  The memory is  set
       to  zero.  If nmemb or size is 0, then calloc() returns either NULL, or
       a unique pointer value that can later be successfully passed to free().

       malloc() allocates size bytes and returns a pointer  to  the  allocated
       memory.   The  memory  is  not  cleared.   If  size is 0, then malloc()
       returns either NULL, or a unique pointer value that can later  be  suc-
       cessfully passed to free().

       free()  frees  the memory space pointed to by ptr, which must have been
       returned by a previous call to malloc(), calloc() or realloc().  Other-
 Manual page malloc(3) line 1
```

Integer Vulnerability: Underflow

Netscape vulnerability

```c
void getComm(unsigned int len, char *src)
{
    unsigned int size;

    size = len - 2;

    char *comm = (char *)malloc(size + 1);
    memcpy(comm, src, size);
    return;
}
```

- $(len - 2)$ can underflow: $1 - 2 = 2^{32} - 1$
- ... so that $(size + 1)$ can overflow: $(2^{32} - 1) + 1 = 0$
- and an attacker may corrupt the heap

Integer Vulnerability

```c
struct dcon_platform_data { ...
    u8 (*read_status)(void);
};
/* ->read_status() implementation */
static u8 dcon_read_status_xo_1_5(void)
{
    if (!dcon_was_irq())
        return -1;
    ...
}
static
struct dcon_platform_data *pdata = ...;
irqreturn_t dcon_interrupt(...)
{
    int status = pdata->read_status();
    if (status == -1)
        return IRQ_NONE;
    ...     }
```

Linux Kernel bug in

OLPC display driver:

What is happening here?

Integer Vulnerability: Sign misrepresentation

```
struct dcon_platform_data { ...
    u8 (*read_status)(void);
};
/* ->read_status() implementation */
static u8 dcon_read_status_xo_1_5(void)
{
    if (!dcon_was_irq())
        return -1;
    ...
}
static
struct dcon_platform_data *pdata = ...;
irqreturn_t dcon_interrupt(...)
{
    int status = pdata->read_status();
    if (status == -1)
        return IRQ_NONE;
    ...    }
```

- `status` can never get negative.
- `read_status` returns $-1 =$ `0xff`
- `0xff` gets zero-extended: `0x000000ff`
- Error handling fails

Integer Vulnerability: Truncation

```
int
detect_attack(u_char *buf, int len,
                      u_char *IV)
{
    static word16 *h =  ...;
    static word16 n = ...;
    word32 l;
    ...
    if (h == NULL) {
        debug("Install crc attack"\
              " detector.");
        n = l;
        h = (word16 *) xmalloc(n *
                          sizeof(word16));
    }
    ...
}
```

- Example: CVE-2001-0144, SSH
- n and l different types
- Assignment $n = l$ could cause a truncation
- Results in exploitable heap corruption

Integer Vulnerabilities: Mitigation

- Correct checks
- Automated testing tools: e.g., KINT
- Use type-safe types
  - SafeInt library (C++)
  - BigInteger (Java)
- Java SE8: Integer Arithmetic Overflow/Underflow Detection API
  - `Math.addExact`, `Math.incrementExact`, ...
  - throws `ArithmeticException`

Integer Vulnerabilities: Summary

Classes of bugs:

- Integer Overflow
- Integer Underflow
- Signedness Error
- Truncation

Exploit

- Denial of Service: Bypass error checking, etc
- Logical Flaw: Bypass Authentication Routine, etc
- Memory Corruption: Often incorrect heap allocation

Heap Corruption

- Heap Overflow
- Use After Free
- Double Free
- Integer error
- Signal Race

Literature / Links

- Marth (2018): Memory Corruption Tutorial. https://www.proggen.org/doku.php?id=security:memory-corruption:start
- Saito (2016): A Survey of Prevention/Mitigation against Memory Corruption Attacks.
- Borisov (2005): Fixing Races for Fun and Profit: How to abuse atime, Usenix Security.
- Cai (2009): Exploiting Unix File-System Races via Complexity Attacks, Oakland.
- Scut (2001): Exploiting Format String Vulnerabilities.
- Google Project Zero: http://googleprojectzero.blogspot.co.at/

Literature / Links

- Brumley (2007): RICH: Automatically Protecting Against Integer-Based Vulnerabilities. NDSS.
- Dietz (2012): Understanding Integer Overflow in C/C++. ICSE.
- Wang (2012): Improving Integer Security for Systems with KINT. OSDI.
- Corelan Team: Exploit writing tutorial part 6 : Bypassing Stack Cookies, SafeSeh, SEHOP, HW DEP and ASLR.
- Meer (2010): Memory Corruption Attacks. The Almost Complete History. BlackHat.

Literature / Links

- Solar Designer (1997): Getting around non-exectuable stack. BuqTraq.
- Nergal (2001): Advanced return-into-lib(c) exploits. Phrack 58-4.
- Shacham (2012): Return-Oriented Programing: Systems, Languages, and Applications
- Shacham (2004): On the effectiveness of Address-Space Randomization.

Literature / Links

- Bulba and Kil3r (2000): Bypassing Stackguard and Stackshield, Phrack 56.
- Richarte (2002): Four different tricks to bypass Stackshield and Stackguard protection.
- Corelan Team: Exploit writing tutorial part 6 : Bypassing Stack Cookies, SafeSeh, SEHOP, HW DEP and ASLR.
- Veen (2017): The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. ACM CCS.

Thank's for your attention!

clemens.hlauschek@inso.tuwien.ac.at

https://security.inso.tuwien.ac.at/