Advanced Security for Systems Engineering – VO 04: Advanced Attacks on Applications 2

Clemens Hlauschek, Christian Brem
Agenda

Stack Smashing and Shell Code writing
  Stack Buffer Overflow
  Writing Shellcode

Stack Smashing Mitigations And Circumvention
  Vulnerable Functions
  Mitigation Techniques
  Circumventing W⊕X
  Defeating ASLR
  Circumventing Stack Canaries

More Classes of Vulnerabilities
  Out-of-bounds Read
  Race Condition
  Format String Vulnerability
  Integer Errors
Stack Smashing and Shell Code writing
Basic stack layout, a horizontal perspective

string grows

buffer

return address

stack grows
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 successful exploitation, it is often necessary to be able to write, debug, and analyze shellcode
- Best to write in assembly
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

```
1     jmp short L1
2     L1:
3     .string '/bin/sh'
```
Shellcode Writing: Example Linux/x86

```
jmp 32                      // relative jump (to line 14)
pop %esi                    // pointer to "/usr/bin/vim" now in %esi
xor %eax, %eax
movb $0x0, 0xc(%esi)        // prepare arguments for sys_execve
mov %esi, 0xd(%esi)
movl $0x0, 0x11(%esi)
mov %esi, %ebx
lea 0xc(%esi), %ecx
lea 0xd(%esi), %edx
movb $0xb, %al             // call to sys_execve via int80
int $0x80
movb $0x1,%al              // call to sys_exit via int80
int $0x80
call −36                  // relative call (line 2), pushes %eip
"/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`, `vprintf` - 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, fgets
- 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

```
%eip = 0xbf86

... sub $400, %esp
    mov $3, %ebx
    xor %eax, %eax
```

```
... ff e3 (jmp *%ebx)
...
```

```
....
argv
argc
```

```
return address 0x0867
```

```
%esp
%ebx
```
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

Circumvention: Return-into-text, Return-into-libc, ROP
Redirect control flow to a useful function in the .text (code-) section.

```bash
$ 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();
gdb> p &hello
$1 = (void *)__NORMAL_CALLFRAME at 0x804848c
```

```c
/* some other * code at addr: * 0x6571 */
int x = 1;
printf("Hello");
```
We can return into any function of any library the process is linked with.
Several functions can be chained together.

Return-into-libc: Function Chaining

buffer fill-up

function1

eplg

arg1

arg2

function2

eplg

arg1

arg2

OFFSET

buffer

return address

add ret

$OFFSET, %esp

stack grows
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
- Automatic increment of %eip
- Change control flow by changing value of %eip
- Sequence of CPU instructions 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

- 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
- %esp is incremented by both the pop and the ret instruction
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 $W \oplus X$, 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.
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 segment, 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.
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
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
What is the problem here?

```c
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

```
string grows

buffer | auth | canary

return address

stack grows
```
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

```c
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 \texttt{str} with the first \texttt{strcpy}. With the second \texttt{strcpy}, attacker can overwrite any address in memory. Patching the \texttt{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.
Frame pointer of previous caller is stored just after return address.
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.
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

<table>
<thead>
<tr>
<th>copy of arguments</th>
<th>char * ptr</th>
<th>int local1</th>
<th>buffer</th>
<th>canary</th>
<th>return address</th>
<th>frame pointer</th>
</tr>
</thead>
</table>

stack grows
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.
Brute forcing 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.
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 Vulnerabilities
Heartbleed

```c
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;
    }
}
```

HeartbeatRequest

01 | length | «length» bytes | e7f0d31...

| type | length | payload | random padding |

02 | length | «length» bytes | dc06848...

HeartbeatResponse
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 comparison 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 setuid-root)

```c
if (access("file", O_RDONLY) != 0)
{
    exit(1);
}
fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

**Unix access syscall**

<table>
<thead>
<tr>
<th>Linux Programmer's Manual ACCESS(2)</th>
<th>Unix access syscall</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NAME</strong></td>
<td>access - check real user's permissions for a file</td>
</tr>
<tr>
<td><strong>SYNOPSIS</strong></td>
<td>#include &lt;unistd.h&gt;</td>
</tr>
<tr>
<td></td>
<td>int access(const char *path, int mode);</td>
</tr>
<tr>
<td><strong>DESCRIPTION</strong></td>
<td>access() checks whether the calling process can access the file pathname. If pathname is a symbolic link, it is dereferenced.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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 easily determine the invoking user's authority.</td>
</tr>
</tbody>
</table>

---

Solved Confused Deputy problem

Introduced Time Of Check To Time Of Use races
Race Condition: access/open TOCTTOU

Victim (installed setuid-root)

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

fd = open("file", O_WRONLY);

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

Attacker

```c
...

// 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)

```c
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)

```c
// 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

- Brute force
- 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

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

Secure?

Countermeasure:

- Decrease counter before checking PIN
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
Recap: Stack Layout, Function Parameters

```c
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

Parameter %n allows write: Control flow hijacking possible

```
$ ./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...
```
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 $\neq$ ($\infty$-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;

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 $a + x \times b > MAX$:
  
  $$x > (MAX - a)/b$$
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

void getComm(unsigned int len, char *src) {
    unsigned int size;
    size = len - 2;
    char *comm = (char *) malloc(size + 1);
    memcpy(comm, src, size);
    return;
}
**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

Linux Kernel bug in OLPC display driver: What is happening here?
### Integer Vulnerability: Sign misrepresentation

```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;

    ...  
}
```

- status can never get negative.
- read_status returns \(-1 = 0xff\)
- \(0xff\) gets zero-extended: \(0x000000ff\)
- Error handling fails
Integer Vulnerability: Truncation

```c
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

- Google Project Zero: http://googleprojectzero.blogspot.co.at/
Literature / Links

■ Dietz (2012): Understanding Integer Overflow in C/C++. ICSE.
■ Corelan Team: Exploit writing tutorial part 6: Bypassing Stack Cookies, SafeSeh, SEHOP, HW DEP and ASLR.
Literature / Links

Literature / Links

- 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/