Exploiting Software Vulnerabilities

Software Vulnerabilities

Integer Overflows and Format String Bugs

ↄ **All wrongs reversed** – under CC-BY-NC-SA 4.0 license

Universidad Zaragoza
1542

Dept. of Computer Science and Systems Engineering
University of Zaragoza, Spain

Course 2021/2022

Master's Degree in Informatics Engineering

University of Zaragoza

Seminar A.25, Ada Byron building

cc BY NC SA

Outline

1. Integer Vulnerabilities
   - Wraparound
   - Overflow
   - Truncation
   - Signedness errors

2. Format String Vulnerability
   - Formatted Output Functions
   - Description of the Vulnerability
   - Mitigation Strategies

Universidad Zaragoza

Outline

Integer Vulnerabilities

Types of integer vulnerabilities

- **Overflow**
  - Occurs at runtime when the result of an integer expression exceeds the maximum value for its respective type
  - For instance, two 8-bit unsigned integers may require up to 16 bits
- **Underflow**
  - Occurs at runtime when the result of an integer expression is less than its minimum value. So it wraps to the maximum integer for the integer type
  - For instance, subtracting $0 - 1$ and storing the result in a 16-bit unsigned integer will result in a value of $2^{16} - 1$ (not -1)
- **Truncation**
  - Occurs when assigning an integer with a greater width to a smaller width
  - For instance, converting an `int` to a `short` discards the leading bits of the `int` value, resulting in a (potential) information loss
- **Signedness error**
  - Occurs when a signed integer is interpreted as unsigned, or vice versa
  - In the two-complement representation, such conversions cause the sign bit to be interpreted as the most significant bit (i.e., $2^{32} - 1 \neq -1$)

Universidad Zaragoza

Integer Vulnerabilities

Examples

```c
struct pixmap {
  unsigned char *p;
  int x;
  /* xsize */
  int y;
  /* ysize */
  int bpp;
};
typedef struct pixmap pix;
......
void readpgm(char *name, pix * p) {
   /* read pgm */...
   pnm_readpaminit(fp, &inpam);
  p->x=inpam.width;
  p->y=inpam.height;
   if(!(p->p=(char *)malloc(p->x*p->y)))
     F1("Error at malloc");
   for(i=0; i<inpam.height; i++){
     pnm_readpamrow(&inpam, tuplerow);
   for(j = 0; j<inpam.width; j++)
     p->p[i*inpam.width+j]=sample;
  }
}
```

```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;
}
```

```c
static inline u32 *decode_fh(u32 *p, struct svc_fh *fhp) {
   int size;
   fh_init(fhp, NFS3_FHSIZE);
   size = ntohl(*p++);
   if (size > NFS3_FHSIZE)
      return NULL;
   memcpy(&fhp->fh_handle.fh_base, p, size);
   fhp->fh_handle.fh_size = size;
   return p + XDR_QUADLEN(size);
}
```

```c
int detect_attack(u_char *buf, int len, u_char *IV){
   static word16 *h = (word16 *) NULL;
   static word16 n = HASH_MIN_ENTRIES;
   register word32 i, j;
   word32 l;
   ...
   for(l=n; l<HASH_FACTOR(len/BSIZE); l=l<<2);
   if (h == NULL) {
      debug("Install crc attack detector.");
      n = l;
      h = (word16 *) xmalloc(n*sizeof(word16));
   } else
      for (c=buf, j=0; c<(buf+len); c+=BSIZE, j++){
         for (i = HASH(c) & (n - 1); h[i] != UNUSED; i = (i + 1) & (n - 1))
            ...;
         h[i] = j;
      }
   }
```

Integer Vulnerabilities

Exploiting integer bugs

Usually, they are indirectly exploited

- **Arbitrary code execution**
  - For instance, insufficient memory allocation exploited by buffer overflows, heap overflows, overwrite attacks, etc.
- **Denial of Service**
  - For instance, excessive memory allocation or infinite loops
- **Attacks to bypass sanitization**
  - For instance, skipping an upper bounds check that ignores unexpected negative integer values
- **Logic errors**
  - For instance, manipulating the reference counter by forcing a referenced object to be freed prematurely

Further reading: *Understanding Integer Overflow in C/C++*. W. Dietz et al. ACM Trans. Softw. Eng. Methodol. 25, 1, Article 2 (December 2015) doi: 10.1145/2743019

Universidad Zaragoza

Integer Vulnerabilities

Consequences

- **Silent break**
  - Compiler optimizations can result in the exploitation of undefined behavior
- **Time bombs**
  - Today works, but improvements in optimization technologies can take advantage of it
- **Illusion of predictability**
  - Some compilers, at some optimization levels, have predictable behavior for some undefined operations
- **Informal dialects**
  - Some compilers have flags to force the two-complement behavior on signed overflows
- **Non-standard standards**
  - Meaning of overflow changes between standards (e.g., `1 << 31`)
    - Implementation defined in ANSI C and C++98
    - Undefined by C99 and C11

Universidad Zaragoza

Integer vulnerabilities
Wraparound – unsigned data types

- **Any calculation involving unsigned operands can never overflow**
- *What if the result of an operation cannot be represented by the resulting unsigned integer type?*
  - **The result is reduced by modulo the number that is one greater than the largest value that can be represented by the resulting type**
- Can occur with **addition and multiplication operations**
  - n-bit addition/subtraction operations require $n + 1$ bits of precision
  - Similarly, multiplying n-bit requires $2n$ bits of precision

Universidad Zaragoza

Integer vulnerabilities

Wraparound – unsigned data types

How to avoid them?

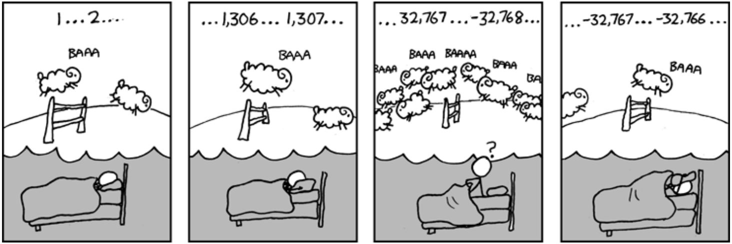
- **Check the wraparound, either before performing the operation that would cause it to occur or after**
- <`limits.h`> limits are useful, but note that **naive use of them does not work**:

```
unsigned i, j, sum;
if (sum + i > UINT_MAX)
    // Too big error
else
    sum += i;
```

Universidad Zaragoza

Integer vulnerabilities

Overflow – signed data types



- **Occurs when a signed integer operation results in a value that cannot be represented in the resulting type**
- **Signed integer overflow is undefined behavior in C, allowing implementations to silently wrap (the most common behavior), trap, or both**
- Since signed integer overflow produces a silent wraparound in most existing C compilers, **some programmers assume that this is a well-defined behavior**

Universidad Zaragoza

Integer vulnerabilities

Find and fix overflows

Shift operations

- **Operand values are checked for limits**
- If the verification passes, then the shift is made

Arithmetic operations

- **Problem: n-bit addition/subtraction operations require $n + 1$ bits of precision**
- Similarly, multiplying n-bit requires $2n$ bits of precision
  - In C, addition, subtraction, multiplication, negation, and division **can result in overflow or underflow**
  - When -2^{n-1} is negated or divided by -1, the result overflows and wraps back to -2^{n-1}
- **Three different methods** can be applied

Universidad Zaragoza

Integer vulnerabilities

Find and fix overflows

Detecting overflow for an operation on two signed integers s_1 and s_2

- **Precondition test**:

$$((s_1 > 0) \land (s_2 > 0) \land (s_1 > (\texttt{INT\_MAX} - s_2))) \lor$$
$$((s_1 < 0) \land (s_2 < 0) \land (s_1 < (\texttt{INT\_MAX} - s_2)))$$

- **Post-condition test of CPU flag** (overflow flag)
- **Post-condition test of width extension**
  - Convert s_1 and s_2 to a broader type
  - Perform the operation
  - Check if the result is within the limits w.r.t. the original (narrower) type

Integer vulnerabilities

Truncation

- **Occurs as a result of an assignment or cast from a type with a larger width to a type with a smaller width**
- **Data may be lost if the value cannot be represented in the resulting type**
  - For instance, adding `c1` and `c2` in the following program fragment produces a value outside the limits of **`unsigned char`**, considering an implementation where **`unsigned char`** is represented using 8 bits ($2^8 - 1 = 255$)

```
unsigned char sum, c1, c2;

c1 = 200;
c2 = 90;
sum = c1 + c2;
```

Universidad Zaragoza

Integer vulnerabilities

Signedness errors

Conversion rules

- **A set of rules that provides a mechanism to produce a common type** when
  - Both operands of a binary operator are balanced with a common type
  - The second and third arguments of the conditional operator (`? :`) are balanced with a common type
- Balancing conversions involve two operands of different types
  - One or both operands can be converted
- **Many operators that accept integer operands perform conversions using the usual arithmetic conversions**, including *, /, %, +, -, <, >, <=, >=, ==, !=, &, ^, |, and the condition operator `? :`

Integer vulnerabilities
Signedness errors

Example of conversion on x86-32

```
unsigned int ui = UINT_MAX;
signed char c = -1;

if (c == ui) {
    printf("-1 = 4,294,967,295?\n");
}
```

Integer vulnerabilities

Signedness errors

Some notes

- **Implicit conversions simplify C programming**
- **<u>Please note</u>: conversions have the potential to lose or misinterpret data**
- Avoid conversions that result in:
  - **Loss of value** (cast to a type where the magnitude of the value cannot be represented)
  - **Loss of sign** (cast from signed type to unsigned type resulting in loss of sign)
- **Conversions of integers to a type with greater range and the same signedness are guaranteed safe for all data values in all compliant implementations**

Outline

Universidad Zaragoza

Formatted output functions

- **Consist of a format string and a variable number of arguments**
  - The format string provides a set of instructions that are interpreted by the formatted output function
- By controlling the content of the format string, **a user can control execution of the formatted output function**

Formatted output functions

- **Consist of a format string and a variable number of arguments**
  - The format string provides a set of instructions that are interpreted by the formatted output function
- By controlling the content of the format string, **a user can control execution of the formatted output function**

Format string

- **Character sequences consisting of ordinary characters (excluding %) and conversion specifiers**
- Ordinary characters **are copied unchanged to the output stream**
- Conversion specifiers **indicate how to convert arguments and write the results to the output stream**
  - Starts with the %, interpreted from left to right

Formatted output functions

Conversion specifiers

```
%[flags] [width] [.precision] [{length-modifier}] conversion-specifier
```

- **Additional arguments are ignored**, if the number of arguments is greater than the conversion specifiers
- **Results are undefined**, if the number of arguments is less than the conversion specifiers

Universidad Zaragoza

Format string vulnerability

- Known since 1999 and exploited since 2000
- Unlike BOF, it is considered a programming error
- Easy to find
- Exploitation techniques are basic

Format string vulnerability

Real examples

Application	Found by	Impact	Years
`wu-ftpd 2.*`	security.is	remote root	> 6
`Linux rpc.statd`	security.is	remote root	> 4
`IRIX telnetd`	LSD	remote root	> 8
`Qualcomm Popper 2.53`	security.is	remote user	> 3
`Apache + PHP3`	security.is	remote user	> 2
`NLS / locale`	CORE SDI	local root	?
`screen`	Jouko Pynnönen	local root	> 5
`BSD chpass`	TESO	local root	?
`OpenBSD fstat`	ktwo	local root	?

Adapted from `https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf`

Universidad Zaragoza

Format string vulnerability

When does it appear?

When user input is included in an ANSI C format function (in part or in full)

Format string vulnerability

When does it appear?

When user input is included in an ANSI C format function (in part or in full)

```
void error(char *s)
{
    fprintf(stderr, s);
}
```

*What if *s is equal to “%s%s%s%s%s%s”?*

Format string vulnerability

When does it appear?

When user input is included in an ANSI C format function (in part or in full)

```
void error(char *s)
{
    fprintf(stderr, s);
}
```

*What if *s is equal to "%s%s%s%s%s%s"?*

- **Program will crash (most likely)**: Denial-of-Service
- **Otherwise, the contents of the stack will be printed**: *privacy issues*

Format string vulnerability
Unsafe functions

- fprintf
- printf
- sprintf
- snprintf
- vfprintf
- vprintf
- vsprintf
- vsnprintf
- setproctitle
- syslog
- Others like err*, verr*, warn*, vwarn*

Format string vulnerability

- **Functionality**
  - Simple conversion from C datatypes to string representation
  - The representation format can be specified
  - The resulting string is processed (e.g., output to stdout, stderr, syslog, etc.)
- **How does a format function work?**
  - **The format string, in fact, controls the behavior of the function**
  - Type of parameters to print
  - **Parameters are stored on the stack (pushed), either directly (value) or indirectly (reference)**
- **The calling function knows how many parameters were pushed, as it has to make stack correction after returning** (by calling convention, more on this later)

Format string vulnerability

Specifier	Output	Example	Passed as
`%d` or `i`	Signed decimal integer	392	value
`%u`	Unsigned decimal integer	7235	value
`%o`	Unsigned octal	610	value
`%x`	Unsigned hexadecimal integer	7fa	value
`%X`	Unsigned hexadecimal integer (uppercase)	7FA	value
`%f`	Decimal floating point, lowercase	392.65	value
`%F`	Decimal floating point, uppercase	392.65	value
`%e`	Scientific notation (mantissa/exponent), lowercase	3.9265e+2	value
`%E`	Scientific notation (mantissa/exponent), uppercase	3.9265E+2	value
`%g`	Use the shortest representation: `%e` or `%f`	392.65	value
`%G`	Use the shortest representation: `%E` or `%F`	392.65	value
`%a`	Hexadecimal floating point, lowercase	-0xc.90fep-2	value
`%A`	Hexadecimal floating point, uppercase	-0XC.90FEP-2	value
`%c`	Character	a	value
`%s`	String of characters	sample	reference
`%p`	Pointer address	b8000000	value
`%n`	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.		reference
`%%`	A % followed by another % character will write a single % to the stream.	%	–

Taken from `http://www.cplusplus.com/reference/cstdio/printf/`

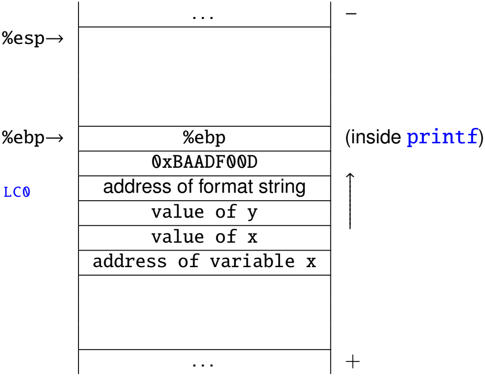
Universidad Zaragoza

Format string vulnerability

Example

```
printf("Values %d, %d, %08x\n", y, x, &x);
do_stuff(); // this call is in 0xBAADF00D address
```

```
_main:
      .....
      mov    eax, DWORD PTR [esp+24]
      lea    edx, [esp+24]
      mov    DWORD PTR [esp+12], edx
      mov    DWORD PTR [esp+8], eax
      mov    eax, DWORD PTR [esp+28]
      mov    DWORD PTR [esp+4], eax
      mov    DWORD PTR [esp], OFFSET FLAT:LC0
      call   _printf
0xBAADF00D:
      call   _do_stuff
      mov    eax, 0
      leave
      ret
```



Format string vulnerability

Channeling problem *When two different types of information channels are merged into one and special escape characters (or sequences) are used to distinguish which channel is currently active*

- One channel is **data channel** (not parsed, just copied): **output strings**
- The other channel is a **control channel: format specifiers**

NOTE: *channeling issues are not security holes,* **but they make the bugs exploitable**

Types

- **Type 1**: format string is partially user-supplied
- **Type 2**: a user-supplied string partially or completely is indirectly passed to a formatted output function

Universidad Zaragoza

Format string vulnerability

Examples by type

Type 1

```
char tmpbuf[512];
snprintf (tmpbuf, sizeof (tmpbuf), "foo: %s", user);
tmpbuf[sizeof (tmpbuf) - 1] = '\0';
syslog (LOG_NOTICE, tmpbuf);
```

Type 2

```
int Error (char *fmt, ...);
...
int someotherfunc (char *user)
{
    ...
        Error(user);
        ...
}
```

Universidad Zaragoza

Format string vulnerability

Feasible attacks

Denial of Service

- **Reading an unallocated memory address crashes the application**

Format string vulnerability

Feasible attacks

Denial of Service

- **Reading an unallocated memory address crashes the application**

Read-what-where

- Interesting format specifiers: `%s`, `%p`
  - **Walk up reading the entire contents of the stack**

```
printf("%s"); // will print the top of the stack
printf("%5$s"); // will print the 5th element of the stack
```

Universidad Zaragoza

 [CC BY-NC-SA 4.0 © R.J. Rodríguez]

Format string vulnerability

Feasible attacks

Denial of Service

- **Reading an unallocated memory address crashes the application**

Read-what-where

- Interesting format specifiers: %s, %p
  - **Walk up reading the entire contents of the stack**

```
printf("%s"); // will print the top of the stack
printf("%5$s"); // will print the 5th element of the stack
```

Write-what-where

- Interesting format specifier: %n
  - Writes to a specified variable **the number of bytes already written**

```
int i;

printf("foobar%n", (int *)&i); // after, i = 6
```

Universidad Zaragoza

Format string vulnerability

Exploitation goals

- **Overwriting the Global Offset Table**
  - Holds the address of each library function used by an ELF program
  - Independent of stack or heap particulars
  - **The control-flow execution is hijacked right after the program invokes the override function**
- **DTORS** (in programs compiled with GNU GCC)
  - Destructor table section (termination routines)
  - Overwrite the pointer to the shellcode and thus **the control-flow execution is hijacked when the program exits**
- **C library hooks**
  - Present in the GNU C library and other proprietary libraries
  - Hooks legitimately used by memory profiling and debugging tools
  - **Control-flow execution is hijacked when the program invokes `malloc`, `realloc`, `realloc`, etc. (before the actual function is run)**
- **`__atexit` structures**
  - Generic handler, **runs when a Linux program invokes `exit`**
- **Function pointers**
  - Commonly used by daemons for command processing or to simulate termination routine handlers

Format string vulnerability

Real example today: 2011 BMW 330i

Associated CVE: CVE-2017-9212 (it also exists in Mercedes-Benz AMG, CVE-2020-16142)

Credits: `https://twitter.com/__0bzy__/status/864704956116254720`

Format string vulnerability
Real example today: Audi A7 2014

Credits: `https://tiger-team-1337.blogspot.com/2020/10/audi-a7-2014-mmi-mishandles-format.html`

Mitigation strategies

Dynamic format strings

- Design your code so that **the user selects from a preexisting format string, rather than incorporating user input directly into the format string**

Byte written restriction

- **Avoid buffer overflows that restrict the number of bytes written by formatted output functions**
- **Use the precision field as part of the** `%s` **conversion specifier**
  - `sprintf(buf, "Wrong command: %s\n", user);` ⇒ `sprintf(buf, "Wrong command: %.495s\n", user);`
  - Even better if you use `snprintf` instead

Mitigation strategies

C11 Annex K functions

- **Provide functions `fprintf_s`, `printf_s`, `snprintf_s`, `sprintf_s`, `vfprintf_s`, `vprintf_s`, `vsnprintf_s`, `vsprintf_s`**
- Differ from their non-_s counterparts by:
  - **Not compatible with the `%n` format conversion specifier**
  - Constraint violation if pointers are null or format string is invalid
- **PLEASE NOTE:** these functions cannot avoid format string vulnerabilities that crash a program or are exploited to view memory
  - That is, **they only prevent from write-what-where attacks**

GNU C compiler flags

- `-Wformat`, `-Wformat-nonliteral`, `-Wformat-security`
  - `-Wformat` checks calls to formatted output functions, examine the format string, and checks that the correct number and types of arguments are supplied
  - `-Wformat-nonliteral` warns if the format string is not a literal string and cannot be verified
  - `-Wformat-security` warns about calls to formatted output function that represent potential security issues

Exploiting Software Vulnerabilities

Software Vulnerabilities

Integer Overflows and Format String Bugs

ⓒ **All wrongs reversed** – under CC-BY-NC-SA 4.0 license

Universidad Zaragoza 1542

Dept. of Computer Science and Systems Engineering
University of Zaragoza, Spain

Course 2021/2022

Master's Degree in Informatics Engineering

University of Zaragoza

Seminar A.25, Ada Byron building

CC BY NC SA