Application Security (apsi)

Lecture at FHNW

Lecture 2, 2021

Arno Wagner, Michael Schläpfer

<arno@wagner.name>, <{michael.schlaepfer, rolf.wagner}@fort-it.ch>

Agenda

More on attack techniques and vulnerabilities

- Buffer Overflow II
- Error handling: The good, the bad and the ugly
- Algorithmic complexity attacks

What do Attackers Want?

- Access to data on the target
- Manipulation of data on the target
- Sabotage of a service on a system (political goals, extortion, vandalism)
- Identity (IP address, DNS name): SPAM, DDoS, jump-off for further attacks
  → The last item is why "hack-back" is a really, really bad idea...
- Resources (RAM, CPU….)
  Newer Trend: Crypto-currency mining

Typical means to get these: Compromise <u>user</u> or system account

Other approaches: Compromise Browser, VM, Sandbox (container, e.g. "Docker") as execution environment
→ Isolation of code may not help! (As long as there is network access...)

Buffer Overflow II

Note: The whole thing is an arms-race. No end is in sight. The defenders are not winning. Robust code (correct and/or redundant) is the only reliable fix.

Now: Buffer overflow used to overwrite variables:

- Also works on heap, in shared-memory, on-disk data-structures, etc.
  → Everywhere where values of variables are stored!

Extensions:

- Code execution via buffer overflow
  Note: May or <u>may not</u> require code injection
  "Return-oriented programming" does not
- Finding buffer-overflow bugs as an attacker via "Fuzzing"

Buffer Overflow II

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

void test (int argc, char *argv[]) {
  char s2[4] = "yes";     // set s2 to "yes"
  char s1[4] = "abc";     // set s1 to "abc"
  strcpy(s1, argv[1]);    // copy argv[1] into s1
  puts(s2);               // print s2
}

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

Demo with GDB: Make input string much larger. What happens?

Buffer Overflow II

Stackframe also contains return address

Note: On x86 this is an absolute address.

- Can be overwritten
  → This usually causes a segfault on function return as most addresses are invalid or non-executable memory.
- Can be overwritten in a targeted fashion
  - Jump to attack-code in same buffer
  - Jumpt to attack-code in a different buffer
  - Jump to some libary-call ("Return-oriented Programming")

Buffer Overflow II

Example OS-Level countermeasure: Memory-layout randomization

Idea:

- Place stack and heap at random places so attacker does not know where the inserted attack code is and hence where to jump

Countermeasure:

1) "NOP-slide"/"Jump catcher": Large area of valid entry-points in attack code
2) Jump to random location

   Note: Catch-area can be made very large

→ Randomization usually only makes the attack somewhat more difficult

Buffer Overflow II

OS-Level countermeasure: NX-Bit

Idea:

- Prevent code execution on the stack, the heap and other non-code areas

Countermeasure: "return-oriented programming"

- Put in jumps to a sequence of legitimate library routines and code fragments
- When one returns, the next one is called
- This techique is generally Turing-complete

→ NX often only helps to make attack more difficult

More detail:
https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf

Attack Technique: Fuzzing

General technique to find buffer-overflows (and other errors)
→ Also useful to test some types of code

Idea:

- Throw random values (unstructured fuzzing) or randomized structured values (structured fuzzing, e.g. valid XML or JSON) at an interface
- Check whether the software crashed or behaves unexpectedly

→ This identifies potential points of attack

Characteristics:

- Easy to automatize
- Finds buffer overflows and bad error handling fast
- Limited power for complex interactions (e.g. login)

Error Handling

Error handling is a vulnerability if the error behavior is problematic

- DoS by resource exhaustion (RAM, CPU, I/O, etc.)
- Data leackage (user names, account numbers, etc. → Lecture 1)
- Leackage of software and OS versions
- Leackage of internals
- Hints that a buffer-overflow or other problem may exist
- Hints to shoddy coding practices
- ...

Case study – Handling too long input

Assume your server gets a query which is too long. What can you do?

- Verbose error
- Generic error
- Silent failure
- Cut it down (dumb or smart)
- Try to accomodate anyways
  - For example: only if space, slow it down, ask for (more) money, ...
- Transform it: Compress, change encoding (picture), remove markup,...
- Ask user to adjust
- Ask user what to do

Remember that the limitations and strategy used will be pretty obvious!

All have serious problems… You still need to select one.

Example Attack: Compression Bomb

A server accepts compressed input and scans it (e.g. Email anti-virus)

- Such data can expand to basically arbitrary length!
- Server cannot drop it (may be important), cannot ask for better input, etc.
- Cannot decompress it before processing, too large

What do do?

Possible solution: Stream-processing

- Decompress incrementally, scan, recompress
- May still take a long time, but no full disks, no exhausted memory, etc.
- It may be acceptable to fail silently after a lot of data (judgment call…)
  But: Email size limits are a problem for large files, e.g. Word documents

Error Handling Practices (1)

Bad practices:

- Fail silently (unless you are really sure it is an attack)
- Give out too little or too much information
- Call debug code in production (oops…)
- Consume a lot of resources
- Log more error-data than you can accomodate
  → Attacker may first flood logs and then experiment unobserved
- Have side-effcts on other activities
- Give misleading information (does not really deter attackers)
- Make it obvious that some errors are not handled (for example by crashing)
- …
- Anything overly complex and violating KISS

Error Handling Practices (2)

Good practices (exceptions may apply, document if you deviate):

- Handle all errors
- Be fault-tolerant whenever it does not cost you much (but prevent DoS)
- Be user-friendly unless that leaks data or can cause DoS
- Do not give details that do not refer to the input
- Do not echo the input (prevent reflector attacks)
- Test error code under high error-load
- Mimimize extra resources used in error handling
- Expect error handling to be abused
- ...
- And always: KISS!

Algorithmic Complexity Attacks

Attack type: DoS (i.e. sabotage), may also be used as supporting attack

- Exploits bad worst-case complexity of algorithms that usually perform well
  - Many algorithms commonly used today have this property!
- Aims at massive slowdown or complete unavailability
  - Note: Unavailability can also be a result of safety mechanisms
    Example: The banking-industry often aborts Mainframe-querys after 500ms
- Effective protection usually exists, but awareness is critical
  … as so often in secure software engineering and secure systems...

Algorithmic Complexity: Sorting

Sorting (the "classical" example):

Quicksort is still commonly used

- It has O(n log n) space (memory) and time complexity on *average*
- It takes O(n^2) space (!) and time in the worst case (space is often on the stack)
- Worst-case is easy to hit: Just feed it pre-sorted data

Fix: Do not use the obsolete Quick-Sort. No, really not!

Use merge-sort or bottom-up heapsort.
=> Minimally slower, but worst-case optimal in time and space.

Algorithmic Complexity: Hashing

Hash-tables are widely used for lookup

- Average case complexity O(1)
- Worst-case complexity: O(n) (time)

Recap: Idea of a hash-table (Note: h is not a cryptograohic hash):

- Store value at a position determined by a hash-function h(value) in an array
  → Hash-function should distribute elements well
- Collision when h(value1) == h(value2): Chain colliding elements in some form
- Attack: Select elements that will cause a lot of collisions

Fix: Add a secret key-quality random element r (>= 64 bit) to each table instance.
Use h(r, value) as hash-funtion.
Also use a modern hash-function like Spooky-Hash or Sip-Hash!

Algorithmic Complexity: Regular Expressions

Regular expression can have exponential execution time (input length)

Attack strategies:

- Use such a string on a bad regular expression (common)
- Supply a bad regular expression (and string to trigger it) to the target (rare)

Simple Example:

- (a+)+ with greedy matching, "aaa...aaaab" as string.

Fix:

- Prevent use of such regular expressions
- Limit runtime and go into error handling if exceeded

Note: Vulnerability depends on matching-engine. NFA-Engines and lazy matching are not vulnerable. These have limitations though, for example no capturing groups in NFA engines.