Application Security (apsi)

Lecture at FHNW

Lecture 2, 2020

Arno Wagner, Michael Schläpfer, Rolf Wagner

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

Agenda

More on attack techniques and vulnerabilities

- Buffer Overflow II
- Code Injection → now only buffer overflow, later (web-sec) more
- Error handling: The good, the bad and the ugly
- Algorithmic complexity attacks

Recap: What do Attackers Want?

- Access to data on the target
- Manipulation of data on the target
- Sabotage of a service (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 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, etc. as execution environment
→ Isolation of code may not help! (As long as there is network access...)

Buffer Overflow II

So far: Buffer overflow to overwrite variables

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

Extension:

- Buffer overflow with code execution
  Note: May or <u>may not</u> require code injection ("return-oriented programming")
- 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?

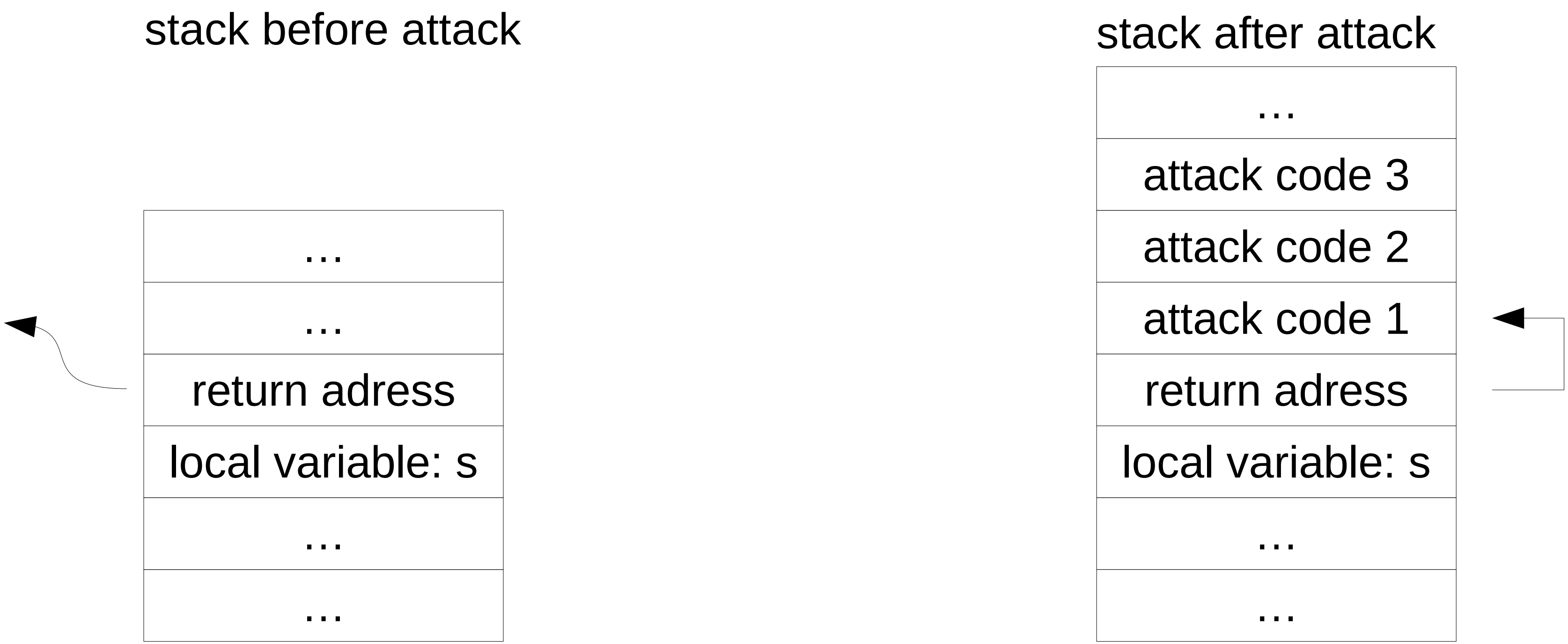
Buffer Overflow II

The stack frame of a running function also contains its return address

Note: On x86 this is an absolute address (other architecures may vary)

- Can be overwritten by a buffer overflow in a local variable
  → This usually causes a segfault on function return
- 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: Scheme (Simple Form)



Overwrite s with: [new value for s][new return address][attack code]

Buffer Overflow II

OS-Level countermeasure: Memory-layout randomization

Idea:

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

Countermeasure:

1) "NOP-slide": Large area of valid entry-points in attack code:
   [nop][nop]…..[nop][attack code]
2) Then overwrite return address with <u>random</u> location

   Note: Catch-area can be made <u>very large</u>

→ Randomization usually only makes the attack somewhat more difficult

Buffer Overflow II

OS-Level countermeasure: NX-Bit

Idea:

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

Countermeasure: "return-oriented programming" (quite complicated)

- 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
- See what happens (Crash? Invalid response? Very slow response?)

→ This identifies potential points of attack

Characteristics:

- Very easy to automatize, various free frameworks exist
- Often finds buffer overflows pretty 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 example)
- Leackage of software and OS versions
- Leackage of internals
- Hints that a buffer-overflow or other problem may exist
- Hints to shoddy coding practices (!)
- Hints that debug code may still be active in a released program
- ...

Error Handling 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 it 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

The limitations of the strategy used will be pretty obvious!
You still need to make a decision.

Case-study: Compression Bomb

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

- Can expand to basically arbitrary length
- Cannot simply 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...)

Error Handling

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 (memory, CPU, I/O bandwith, ...)
- Log more error-data than you can accomodate
  → Attacker may first flood the logs and then experiment unobserved
- Have side-effcts on other activities
- Give misleading information (does not really deter attackers)
- Make it obvious some errors are not handled (for example by crashing)
- …
- Anything overly complex

Error Handling

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 attacked
- ...
- And always: KISS!

Algorithmic Complexity Attacks

Attack type: DoS (may be used to facitilate other attacks)

- Exploits bad worst-case complexity of algorithms commonly used
- Aims at massive slowdown or complete unavailability
  - Unavailability can be the result of maximum execution times
    Example: The banking-industry often aborts Mainframe-querys after 500ms
- Effective protection usually exists, but <u>awareness is critical</u>

Examples:

- Sorting (Quicksort)
- Hash-Tables
- Regular Expressions
- ...

Algorithmic Complexity: Sorting

Sorting (the "classical" example):

Quicksort unfortunately is still commonly used

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

Fix:

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

Algortithmic Complexity: Hashing

Hash-tables are widely used for lookup

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

Recap: Idea of a hash-table:

- 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: Make all elements collide with each other => lookup takes O(n)

Fix: Add random secret element r to each table instance.
Then use h(r, value) as hash-funtion. "r" should be 64 bits or more.

Algorithmic Complexity: Regular Expressions

Regular expression can take exponential execution time (in the input).

Attack possibilities:

- Use an attack string on a bad regular expression in a software (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 vulnerable 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.