Application Security (apsi)

Lecture at FHNW

Lecture 5, 2021

Arno Wagner, Michael Schläpfer

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

Dates

On-Site Lectures (there are no others, MSP will be on-site):

- 22.11.2021: Semester-Exam
  13:15-15:00, Aula, 3.-111
- 6.12.2021: Exam discussion and handing it back + lecture
  12:15-14:45, Aula, 3.-111

Agenda

- Security Testing
- Code Analysis for Security
- Backdoors in Software
- Fuzz-Testing
- Economic Aspects of Security

Security Testing (for Software)

- Pen-Test: Attempt to break in
  - Long catalog of things to try…, for example Fuzzing, injection, default credentials...
- Load-test: Determine high-load behavior
  Also specifically for logging and security mechanisms!
- Code Inspection and review
- Code scanners (for example Fortify)
- Code emulation environments (for example Valgrind or Qemu)

Pen-Test

There are 3 (main) classes:

1) White Box: Tester has credentials (passwords), documentation, maybe even debug access
2) Black Box: Tester is given minimal information (target IP range) not more
3) Grey Box: Somewhere between White Box and Black Box

- Black Box is mostly useless, except as exposure-test
- Typical situation is Grey Box, often because only limited information available
- Sometimes Pen-Test may only be run against test-environments
- Often, Pen-Tests may not do flooding and must be done in off-hours
- Pen-Rests can pretty much always break the target system, no matter how carefully done

How does a Pen-Test fail?

It fails to successfully attack the system!

What do you know in that case?

- Nothing!
  → The customer feels secure, but the wrong things may have been tested
  → Risk is highest in a Black Box test

The customer just fixes the observed issues, nothing else

- Pen-Tests can sometimes identify root-causes – these need to be fixed
  → Example: Unpatched software indicates broken software maintenance
- Pen-Tests are not complete tests (due to time/budget/skill-restrictions)
- Pen-Tests are useful to create awareness

Fuzz-Testing: The Idea

1. Throw (more or less) random values at an interface
   "Interface" can be:
   Server port (web, telnet, …), function call (also library), file, etc.

2. Observer whether behavior deviates from the expected:
   - crash
   - nonsensical return values, including format violations
   - errors that do not fit
   - much longer time to respond (e.g. for finding DoS vulnerabilities)

3. If behavior is non-standard, investigate in detail:
   - "cash" often indicates a buffer overflow
   - "nonsensical return value" and "errors that do not fit"
     often indicate implementation flaws or broken input validation
   - etc.

Possibilities and Limitations

Attacker (wants to find ways to break in):

- ++ Can be automated, can also be done large-scale (scan whole Internet)
- ++ Can also be used against compiled code, in particular programs or libraries where no source code is available to the attacker
- ++ Does allow early estimation whether investing more effort makes sense
- - - Needs significant set-up effort (but may be reusable)
- - - Crashes, etc. may alert the target.

Defender (i.e. developer, integrator, system admin, pen-tester, etc.):

- Basically the same as Attacker

Structured Fuzzing vs. Random Fuzzing

- Random fuzzing does not respect the structure of the expected input data
  - ++ Easier to do
  - - - May not even reach interesting cases
- Structured fuzzing structures the input data to some degree

Example against commandline:

Random: ls <random>

Structured: ls -a <random> … -z <random> <random_optional>
where "-a" .. "-z" are random known allowed options in random order

Example against cgi-script:

Random: http://a.b.c/d/e/f/test.cgi?<random>

Structured: http://a.b.c/d/e/f/test.cgi?a=<random>&...&z=<random>
where "-a" .. "-z" are random known allowed parameters in random order
and the allowed char-set for parameters is respected

Fuzzing for Regular Tests

This is fuzzing <u>not</u> a (real or simulated) attacker

- Does work well for testing data-structure implementations
  Structured fuzzing
  → Should be used to compare against a 2nd (simple, slow) implementation
  This way, it becomes an instance of diversification!

  Examples:
  - Search-tree (test-object) vs. linear search on an array (reference)
  - Karatzuba-Multiplication v.s regular multiplication
  - Priority Queue (Heap) vs. linear search for the largest element

- Does work somewhat for testing of input validation
  → Structured and unstructured, depending on complexity of input

- Does not really work for complex interaction (DB access, business-logic, etc.)
  Counterexamples exist, YMMV.

Useful Fuzzing Tools

- American Fuzzy Lop
  https://en.wikipedia.org/wiki/American_fuzzy_lop_(fuzzer)
- Many more on https://blackarch.org/fuzzer.html
- Some on https://www.owasp.org/index.php/Fuzzing

Many more exist...

Code Inspection and Review

A second person looks at the code and looks for problems

- Limited usefulness if done internally
- Critically dependent on reviewer skill and available time

Direct results (somewhat useful):

- Bugs

Indirect results (very useful, but politically problematic):

- General code quality
- Interface quality
- Skill-level of original coder
- ...

Code Scanners ("Automated Code Review")

A code scanner is a tool that looks for problematic code

- Can be simple (structural) pattern matching
  Example finding "if (a=b) {…}" (C code) and the like
- Can be very sophisticated
  - Data-flow techniques (similar to taint-checking)
  - Check whether input values were looked at at all
  - Memory-leak, use-after-free, etc.
  - …
- Not really easy to use (depends) and may create false sense of security
- May collide with coder ego….

Execution Emulators

Example 1: Valgrind

- Executes code in symbolic form
- Uses JIT and other optimization techniques
- Most useful as memory-debugger (overflows, memory-leakage, …)
- Around 20-25% of original execution speed

Example 2: Qemu

- Software virtualization tool
- Extended debugging options
- Allows non-native configurations (different CPU, etc.)
- Pretty slow...

Patching: Problems with Patching

- Patches may break existing functionality
  - User may not want automated patching (especially enterprise users)
    → Users may forget or delay manual patching (especially home users)
- Patches may introduce new problems or fail to fix existing ones
  - Patches may allow attacks by the software Vendor (and others)!
- Patching needs to be done securely (signature checking, etc.)
  - Many users do not know how to do that
  - Not all automatic patching does this or does it wrongly
- Patches may change operational characteristics
  - Memory/CPU consumption, I/O characteristics, etc.
- Patches may make systems unavailable for a while
  - Patching can take time, sometimes a long time

Problems with not Patching

- Vulnerabilities can accumulate
- Vulnerabilities can become generally known
  - Black-hat hackers analyze patches in order to build attacks
  - Exploit code can become publicly available

- "Emergent Properties": The system may become vulnerable to actual attacks from vulnerabilities that individually cannot (easily) be attacked.

  => If each present vulnerability cannot be attacked individually, a combination may still allow an attack!

Problems with Unavailability of Patches

Why would patches be unavailable?

- The product is EoL
- There is no patching possibility (worst case)
- Nobody that can cares to provide patches
- Patches are delayed long enough to give atackers plenty of time
- Patches <u>are</u> available, but break needed functionality
- Patches <u>are</u> available, but introduce known vulnerabilities

Attackers monitor this situation and may intensify efforts when a system is known to not get patches!

Backdoors

A "Backdoor" is a <u>maliciously</u> placed access that breaks security
It does not matter <u>who</u> places or mandates the backdoor!

- Finding of backdoors
  - In code
  - In libraries, run-time environments, containers, VM images machines, virtualization software,…
  - In cryptography
    → This is impossible for modern "NOBUS" backdoors, but the possibility ("compromised design"/"compromised algorithm") may be identifiable
- Placed by:
  - Disgruntled employees (defense: keep your employees happy…)
  - State-sponsored sabotage
  - Vendors that want marketing data without asking/telling the user

Backdoors in Code

Definition

- A remotely or locally accessible (hidden) undocumented functionality (often a hidden interface) that gives an attacker that knows about it more access than the owner of the software is aware or has authorized.

  Note: This does include bypassing data-access restrictions

Additional characteristics

- May be disguised as coding mistake
- Intentionally made hard to detect on code-level

Backdoor Hiding Techniques

- Meta-technique: Use what appears to be common coding errors
  → If you find it, you do not know whether this was an attack!
  (Remember that coders can be arbitrarily incompetent these days...)
- Use debug code that was "accidentally" left active
- Omit workarounds for known vulnerabilities or implement them only partially
- Use violations of "least surprise" in libraries and system calls or services
  - Use obscure and complex library functionality
- Use race-conditions
- Use bad initialization
- Use intentionally bad or misleading code comments
  → Works well for interface specifications of complex functions
- Use Intentionally obfuscated, complex and/or badly structured code
- Use low-skill coders and then look for vulnerabilities in their code

Finding Backdoors

Using tools (Security scanners):

- Only work if the attacker has not tested against them (many do...)
- Only work if code does not give lots of errors ("hiding a tree in the forest...")

Manually:

- Identify all input from outside and follow the respective data-paths
- Look for functionality that "does not make sense", like very awkward code, unnecessary complex code, complex libraries that are not really used, etc…
- Look for misleading comments and comments that do not make sense
- Verify <u>all</u> functionality

If done right, this is generally <u>more expensive</u> than rewriting the software!

Backdoors in Cryptography

Relatively new trend: "NOBUS" backdoors

- NOBUS = "NObody But US"
- Uses cryptographic properties to protect the backdoor
- Is not distinguishable from secure version without a secret key

=> Only the attacker can see them

This is a "mathematically compromised design"

How do deal with it?

- Assume if the possibility is there, then it is used!
- The secret protecting the backdoor may leak.
  => <u>Do not trust these under any circumstances!</u>

Example 1: ECC

ECC (Elliptic Curve Cryptography) relies on a selected curve

- Generation of a curve can be done in a way to include a backdoor
  → Attacker generates curve, publishes it
  → Attacker has secret knowledge of curve property that facilitates attack

Defense:

- Use only curves that are generated by an "obviously" not compromised procedure and generated by a trusted party
  → Example: Curve25519 Reference: https://safecurves.cr.yp.to/

Example: Dual_EC_DRBG

- Uses a curve that "fell from the sky" (Well, the sky over Fort Meade...)
- Demonstrated to be vulnerable with other, specifically generated curve

Advice: If in doubt, do not use ECC at this time

Example 2: Specially selected Primes for Discreet Logarithms

Idea:

- Select a Prime P so that the discrete logarithm over the generated finite Field is easy to compute
- Detecting this "trapdoor" if P is unknown is likely computationally infeasible

Protection:

- Use your own prime(s)
  → Use methods where each party generates their own primes

Example: http://eprint.iacr.org/2016/961

Economic Aspects of Security

Scope:

- Commercial software (COTS)
- Custom-software (self-built or built-to-order)
- Not in scope: FOSS (that one is more difficult) unless commercially used

Key-Question: Is there profit in insecure software?

- Yes, there is!

Does it Pay to make Software Secure?

Question: Cost of attack * frequency vs. cost of making software secure

- Unfortunately, being insecure (seems) often cheaper
- Attacks are not a major cost-factor:
  https://www.schneier.com/blog/archives/2016/09/the_cost_of_cyb.html
  - Survey over 12,000 incident reports 2004-2014
  - Cost per attack: Average $200'000 which is only about 0.4% of annual revenue
    For comparison: Fraud is around 6% of annual revenue
    → Not seen as a major issue
- Preventing attacks is expensive

Sometimes this calculation fails (but mostly people seem not to care...)

- Example: The offer for Yahoo shrunk from 4.8 billion to 3.8 billion once their >1Million customer records breach became known.

We are missing "Reference-Catastrophes" that normal people understand

So, what to do?

This cannot go on for much longer (or can it?)

- Make the coders liable for insecure code?
  - We do not even have standards who is allowed to write critical code!
- Implement working product liability and require insurance
  - May work in certain markets...but what standards to apply?
    Note: Other engineering disciplines have this!
- Require certifications
  - Well, again to what standards? Industrial certifications are often not worth much.
- Require independent reviews
  - Helps to some degree, but these are expensive, so people try to do without
- Improve CS education to teach IT Security as mandatory topic
  - Not generally done, even today. Why is that? Is the discipline too young?

→ Expect this to be a topic that will remain unsolved for a lont time