

Defence in Depth

Petr Ročkai

Overview

- Part 1: Layered Security
- Part 2: Code Review & Open Design
- Part 3: Mitigation Techniques
- Part 4: Dropping and Separating Privileges
- Part 5: Related Issues

Part 1: Layered Security

Goal: Secure Systems

- no privilege or **access violations**
- no **leaks** of private data
- no unauthorised **resource abuse**
- **availability** of service

Solution: Write Bulletproof Code

- **never works** in practice
- but see also **seL4**

Alternative Solution

- write **good**, even if imperfect, **code**
- keep it **simple**
- use **established components** / libraries
- code **reviews** (both security and correctness)
- **mitigation** techniques (ASLR, Stack Guard, ...)
- least privilege & privilege separation
- minimise inter-component **trust**

Layered Security

- secure each component / layer separately
- many fences: make life hard for the attacker
- log all suspicious failures in your programs

Rules

- if you **detect** an attack **early**, you **win**
 - before anything of value is stolen or compromised
 - if the attacker gives up you also win
- if you win, it doesn't matter how
 - how many holes the attacker punched in your defence

Why Many Layers

- each layer **slows the attacker** down
- each layer has a chance to **detect and report** the attack
- the attacker may **fail to penetrate** further at any point
- obstacles → frustration → mistakes
- more attacker mistakes = better chance that you win

Layering Example

- you run a C program & it was **reviewed** for security
- but a tricky buffer overflow **slipped past**

- the attacker discovers the overflow
- they attempt an exploit, but you use **stack guard**
- the program crashes, **alerting the sysadmin**

- the system goes into **lockdown**
- the buffer overflow is identified and **fixed**
- **you win**

Single Points of Failure

- certain SPOFs are **unavoidable**
- prime example: **the user**
- common failure modes can be **mitigated**
- **bad passwords** × **2FA**
- social engineering × least privilege & strict protocols

- **bad** mitigation: password policies

Part 2: Code Review & Open Design

Code Review

- the practice of reading and **understanding** code
- done by yourself, your team-mates, an external audit
- catches the most egregious **security violations**
- **not** a **foolproof** method
- the law of diminishing returns applies

Code Review: Open Source

- with enough eyeballs, all bugs are shallow
- sounds nice, but is **not true**
- counterexamples: **heartbleed**, **shellshock**, ...
- still very helpful

Security by Obscurity

- the polar opposite of open source
- keep the design secret
- might use proprietary encryption
- keep the source code secret
- obfuscate binaries &c.

Does Not Work

Insecurity by Obscurity

- rarely, if ever, independently reviewed
- the only interested party is the attacker
- often riddled with basic flaws and inadequate crypto
- attackers are often good at reverse engineering
 - disassemblers, debuggers and emulators
 - decompilers and automated control flow analysis
- insider attacks are a thing

Insecurity by Obscurity: Famous Examples

- GSM encryption (A5/1)
 - also an example of intentionally weakened crypto
 - and a practical downgrade attack
- MS Wireless Keyboard (XOR the MAC, CVE-2010-1184)
- MIFARE Classic (reverse engineered & found vulnerable)
- car remotes (Keelog, VW, ...)
- ~ every **copy protection** / DRM scheme ever

Obscurity Benefits

- obscurity could also work in your favour
- think non-updateable software in **tamper-proof** boxes
- hire **expert** programmers & reviewers
- stick with **established crypto**
- contract a few security labs for **external review**

Compromise: Open Design

- you may have reasons to avoid opening your source
- you can still document and **open the design**
- this allows beneficial **independent review**

Use Established Modules

- use **standard**, tested and widely deployed components
 - **especially** for cryptography
- use **standard protocols**, formats &c.
- they had a lot more review than your code
- **never** implement your **own cryptography**
 - implementation bugs are common
 - especially **side channels**
 - sources of randomness are a serious problem

Part 3: Mitigation Techniques

Mitigation

- assumption: **bugs are inevitable**
- idea: make them **hard** or impossible **to exploit**
- **not** a substitute for good code
- part of a **layered security** approach

Mitigation Approaches

- make common bugs **harder to exploit**
- **isolate** components from each other
- principle of **least privilege**
- keep each component **simple**
- **fail securely** whenever possible

Exploit Mitigations

- **W^X** – write XOR execute
- address space layout **randomisation**
- boot-time library **randomised relinking**
- trap sleds (as opposed to nop sleds)
- guard pages
- **malloc** & **mmap** randomisation
- **secure randomness** by default

Isolation: Motivation

- stop **propagation** of faults
- **protect** unrelated applications
- make attacks harder to conduct

Isolation: Approaches

- separate processes
- separate user accounts
- lightweight containers (freebsd jails, linux lxc)
- virtual machines
- physical separation

Sandboxing

- further restrict **dangerous code**
- SELinux, AppArmor (Linux)
- pledge (OpenBSD), capsicum (FreeBSD)
- Chromium content processes (also Edge, also Safari)
- ZeroVM

Isolation Failures

- hyper-threading (SMT) **side channels** (CVE-2005-0109)
- rowhammer (CVE-2015-0565)
- MMU side channel attack (defeats ASLR, CVE-2017-5925)

Isolation: Not Applicable

- how do you protect the database from wordpress?
- bookmarks, cookies or history from the browser?

Simplicity

- **complex** code often has **more bugs**
- **simpler** code means **fewer bugs**
- applies to **design** as well
- keep the code clean and **readable**
- avoid clever hacks and dubious optimisation
- resist adding unnecessary features

Minimise Trust

- trust is the opposite of isolation
- servers should **not trust** clients & vice versa
- **never trust** your **inputs** (see previous lectures)
- do not trust the **network**
- never run unsigned code
- **faults propagate** along trusted channels

Fail Safe vs Fail Secure

- behaviour during **failure** is often very important
- fail **safe**: do not endanger lives or property
- fail **secure**: ensure security is not broken
- **not** an either-or choice
- but not completely orthogonal either

Compare:

```
if ( check_access() == EDENIED )  
    die( "access denied" );
```

with

```
if ( check_access() != OK )  
    die( "access denied" );
```

What happens if `check_access()` returns `ENOMEM`?

Errors are Hard to Test

- **error paths** often contain vulnerabilities
- often inadequately tested
- use automated tools (fuzzing, static analysis)

Errors are Info Leaks

- stack traces, database details
- the **padding oracle** attack

Part 4: Dropping & Separating Privileges

Principle of Least Privilege

- give only privilege **required** to get the job done
- applies to both programs and users
- does **not** prevent security holes
- this is again a mitigation technique
- Saltzer & Schröder 1975

Dropping Privileges

- how to get rid of excessive privilege?
- use dedicated, restricted user accounts
- chroot jailing to restrict file system access
- **sandboxing** (selinux, pledge, ...)

Privilege Drop: Common Example

- opening ports below 1024 requires **root**
- so does reading SSL **private keys**
- nothing much else in httpd does, though
- after startup, drop to an **unprivileged user**
- maybe also lock out filesystem with **chroot**

Trusted Computing Base

- the entirety of **code important for security**
- includes most **application software**
- capable of violating user's **security constraints**
- **should be as small** as possible
- **usually** very **large** in practice
- sufficiently sandboxed code is not part of the TCB

Privilege Separation

- multiple processes
- separate responsibilities
- simple & robust inter-process protocol
- **more powerful** than the least privilege approach
- capable of removing code from the TCB

OpenSSH

- all **network code** runs in a separate process
- under a special user & chrooted
- privileged process is well isolated
- the latter decides everything security-relevant

Other Examples

- mail software: qmail, postfix
- OpenBSD relayd, httpd, bgpd, ntpd, ...
- chromium (see also sandboxing)

Part 5: Related Issues

Programming Languages

- not all languages are equal from security POV
- C is among the **worst** options
- C++ is **better** if used correctly
- Java is **better** yet (memory safe)
- yet safer languages exist (Rust, Haskell, ...)

Keep Yourself Informed

- what is the **security record** of the components you use?
- learn from your **mistakes**
- or even better, from **mistakes of others**
- learn about the latest trends
- read security blogs, papers, ...

Security Patterns

- like software design patterns (Gang of Four)
- commonly used designs and techniques
- recommended as good design by multiple sources

<http://www.munawarhafiz.com/securitypatterncatalog/index.php>

Summary

- **never assume** your code is perfect
- every defence could (and will) **fail**
- always stack **multiple approaches**
- be prepared for the **worst case**

Questions?