CS-412 Software Security

Introduction

Mathias Payer

EPFL, Spring 2019

Course outline

- Secure software lifecycle
- Security policies
- Attack vectors
- Defense strategies: mitigations and testing
- Case studies: browser/web/mobile security

Hack the planet!

Figure 1:

About me

- Instructor: Mathias Payer
- Research area: software/system security
  - Memory/type safety
  - Mitigating control-flow hijacking
  - Compiler-based defenses
  - Binary analysis and reverse engineering
- Avid CTF player (come join the polygl0ts)
- Homepage: `http://nebelwelt.net`

Semester and master projects

- Interested in security?
- We supervise projects in software security!
  - Software testing: fuzzing
  - Software testing: sanitization
  - Mitigation
  - Program analysis
- Ping me if interested

Figure 2:

Internship/remote master project

- Internship number 24346
- "TLS Certificate analyser"
- Who? DDPS, Marc Doudiet
- Ping me if interested

Why should you care?

- Security impacts everybody's day-to-day life
- Security impacts your day-to-day life
- User: make safe decisions
- Developer: design and build secure systems
- Researcher: identify flaws, propose mitigations

Morris Worm

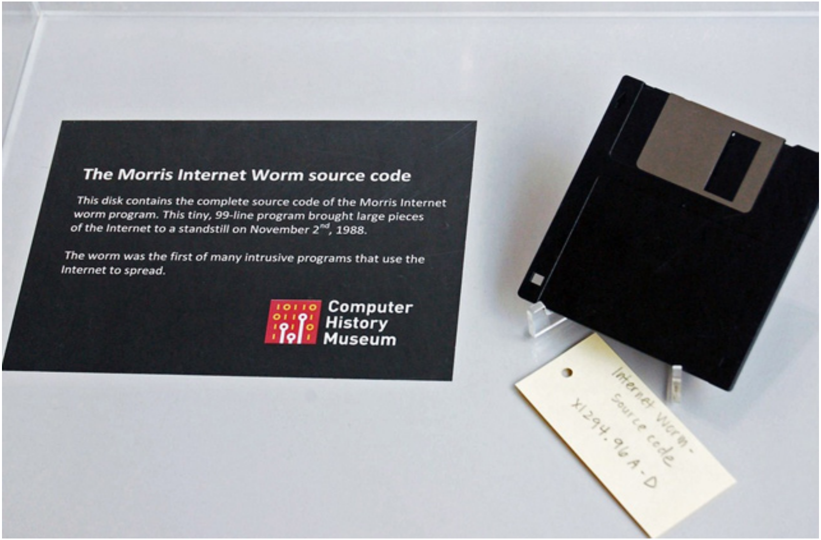


Figure 3

Morris Worm: What it did

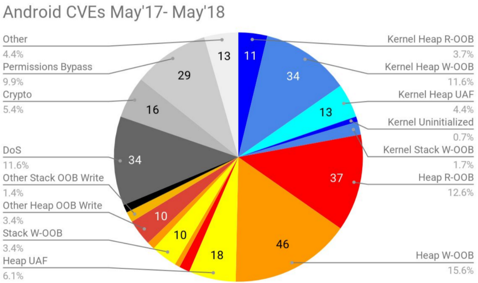
- Brought down most of the internet in 2nd November, 1988
  - Buffer overflow in `fingerd`, injected shellcode and commands.
  - Debug mode in `sendmail` to execute arbitrary commands.
  - Dictionary attack with frequently used usernames/passwords.
- Buggy worm: the routine that detected if a system was already infected was faulty and the worm kept reinfecting the same machines until they died.
- Reverse engineering of the worm

C and C++ are unsafe. Humans too.

- Kostya Serebryany, Making C/C++ safer
- Lots of scary bugs with scary names and logos
- C and C++ are neither memory nor type safe
  - Root causes: read/write out-of-bounds (OOB) or after-free (UAF), integer overflow, type confusion, ...
  - Consequences: (remote) code execution, information leak, privilege escalation, safety/reliability issues, ...

MT covers >60% CVEs

Android CVEs (*)



(*) Source: High/Critical CVEs, May 2017- May 2018

Figure 4:

Google Chrome

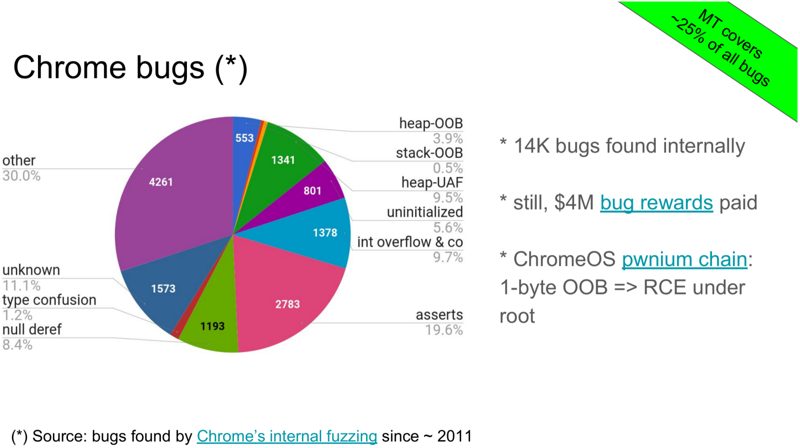


Figure 5:

Low-level software is highly complex

- Low-level languages (C/C++) trade type safety and memory safety for performance
- Google Chrome: 76 MLoC
- Gnome: 9 MLoC
- Xorg: 1 MLoC
- glibc: 2 MLoC
- Linux kernel: 17 MLoC

Software complexity (1/2)

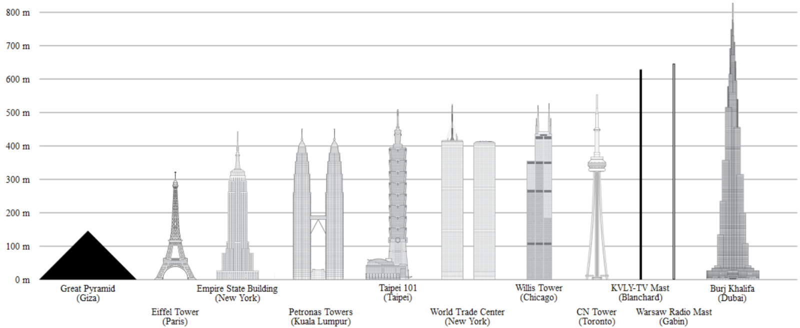


Figure 6:

Software complexity (2/2)

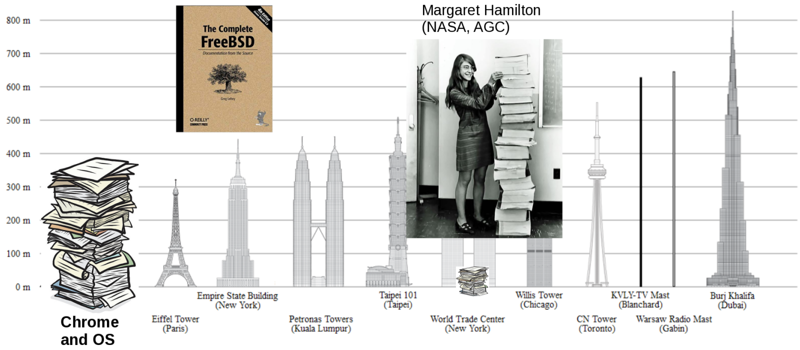


Figure 7:

~100 mLoC, 27 lines/page, 0.1mm/page equals roughly 370m

Software Engineering versus Security

Software engineering aims for

- **Dependability:** producing fault-free software
- **Productivity:** deliver on time, within budget
- **Usability:** satisfy a client's needs
- **Maintainability:** extensible when needs change

Software engineering combines aspects of PL, networking, project management, economics, etc.
Security is secondary and often limited to testing.

Definition: *Security*

Security is the application and enforcement of policies through mechanisms over data and resources.

- Policies specify what we want to enforce
- Mechanisms specify how we enforce the policy (i.e., an implementation/instance of a policy).

Security best practices

- Always lock your screen (on mobile/desktop)
- Unique password for each service
- Two-factor authentication
- Encrypt your transport layer (TLS)
- Encrypt your messages (GPG)
- Encrypt your filesystem (DM-Crypt)
- Disable password login on SSH
- Open (unkown) executables/documents in an isolated environment

Definition: *Software Security*

Software Security is the area of Computer Science that focuses on (i) testing, (ii) evaluating, (iii) improving, (iv) enforcing, and (v) proving the security of software.

Why is software security difficult?

- Human factor (programmer, software architect, . . .)
- Concept of weakest link
- Performance
- Usability
- Lack of resources (time, money)

Software security best practices?

- Properly design software
- Clear documentation (design and implementation)
- Leverage frameworks (don't reimplement functionality)
- Code reviews
- Add rigorous security tests to unit tests
- Formal verification for components that can be verified (protocols, small pieces of software)
- Red team software
- Offer bug bounties

Definition: *Software Bug*

A software bug is an error, flaw, failure, or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways. Bugs arise from mistakes made by people in either a program's source code or its design, in frameworks and operating systems, and by compilers.

Source: Wikipedia

Common bugs: spatial memory safety violation

```
void vuln() {
  char buf[12];
  char *ptr = buf[11];
  *ptr++ =  10;
  *ptr = 42;
}
```

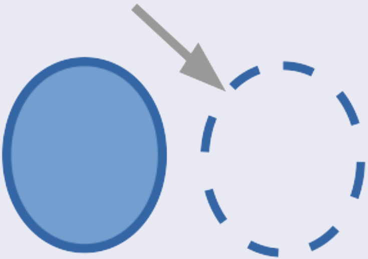


Figure 8:

Common bugs: temporal memory safety violation

```
void vuln(char *buf) {
  free(buf);
  buf[12] = 42;
}
```

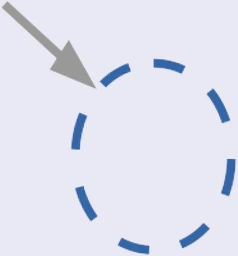


Figure 9:

Common bugs: type confusion

```
class Base {};
class Greeter : Base {};
class Exec : Base {};
Greeter *g = new Greeter();
Base *b = static_cast<Base*>(g);
Exec *e = static_cast<Exec*>(b);
...
```

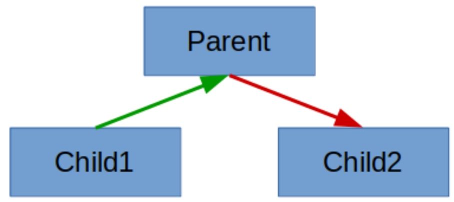


Figure 10:

Definition: *Software Vulnerability*

A vulnerability is a software weakness that allows an attacker to exploit a software bug. A vulnerability requires three key components (i) system is susceptible to flaw, (ii) adversary has access to the flaw (e.g., through information flow), and (iii) adversary has capability to exploit the flaw.

Problem: broken abstractions

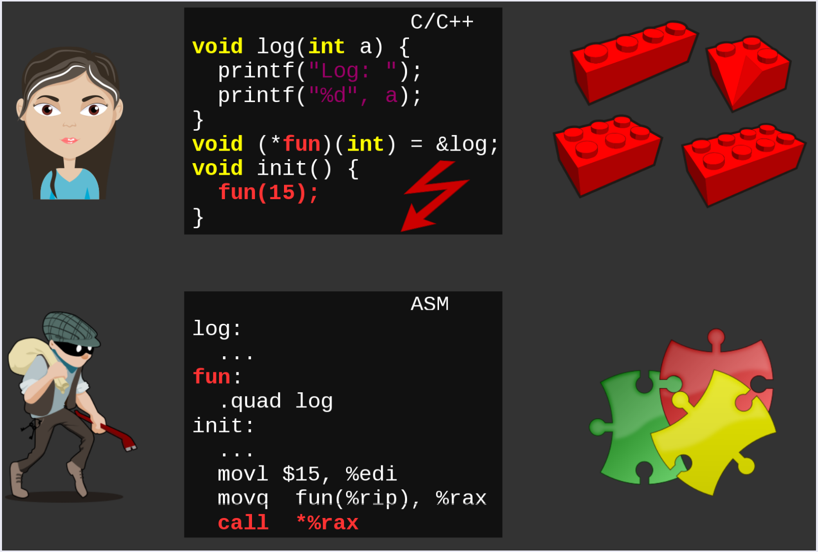


Figure 11:

Course goals

Software running on current systems is exploited by attackers despite many deployed defense mechanisms and best practices for developing new software.

> *Goal: understand state-of-the-art software attacks/defenses across all layers of abstraction: from programming languages, compilers, runtime systems to the CPU, ISA, and operating system.*

Learning outcomes

- Understand causes of common weaknesses.
- Identify security threats, risks, and attack vector.
- Reason how such problems can be avoided.
- Evaluate and assess current security best practices and defense mechanisms for current systems.
- Become aware of limitations of existing defense mechanisms and how to avoid them.
- Identify security problems in source code and binaries, assess the associated risks, and reason about severity and exploitability.
- Assess the security of given source code.

Syllabus: Basics

- **Secure software lifecycle:** Design; Implementation; Testing; Updates and patching
- **Basic security principles:** Threat model; Confidentiality, Integrity, Availability; Least privileges; Privilege separation; Privileged execution; Process abstraction; Containers; Capabilities
- **Reverse engineering:** From source to binary; Process memory layout; Assembly programming; Binary format (ELF)

Syllabus: Policies and Attacks

- **Security policies:** Compartmentalization; Isolation; Memory safety; Type safety
- **Bug, a violation of a security policy:** Arbitrary read; Arbitrary write; Buffer overflow; Format string bug; TOCTTOU
- **Attack vectors:** Confused deputy; Control-flow hijacking; Code injection; Code reuse; Information leakage;

Syllabus: Defenses

- **Mitigations:** Address Space Layout Randomization; Data Execution Prevention; Stack canaries; Shadow stacks; Control-Flow Integrity; Sandboxing; Software-based fault isolation
- **Testing:** Test-driven development; Beta testing; Unit tests; Static analysis; Fuzz testing; Symbolic execution; Formal verification
- **Sanitizer:** Address Sanitizer; Valgrind memory checker; Undefined Behavior Sanitizer; Type Sanitization (HexType)

Syllabus: Case studies

- **Browser security:** Browser security model; Adversarial computation; Protecting JIT code; Browser testing
- **Web security:** Web frameworks; Command injection; Cross-site scripting; SQL injection
- **Mobile security:** Android market; Permission model; Update mechanism

Course material

- Slides/homepage:
  `https://nebelwelt.net/teaching/19-412-SoSe/`
- Text book: Mathias Payer, Software Security: Principles, Policies, and Protection
- Moodle for discussions
- Complementing books
  - Trent Jaeger, Operating System Security
  - Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Operating Systems: Three Easy Pieces
- Labs and exercises

Text book: SS3P

- Software Security: Principles, Policies, and Protection
  - There were no text books when I started developing this class.
  - There will be continuous updates, don't print it (yet).
  - Feedback is encouraged: let me know if you find issues, missing information, lack of context, or typos.
- Main Topics
  - Software and System Security Principles
  - Secure Software Life Cycle
  - Memory and Type Safety
  - Defense Strategies
  - Attack Vectors
  - Case Studies: Mobile and Web

SS3P: Software and System Security Principles

- Basic security properties
- Assessing the security of a system
- Confidentiality, Integrity, and Availability
- Isolation, Least Privilege, Compartmentalization
- Threat Modeling

Secure Software Life Cycle

- Integration of security into design
- Continuously assess security during implementation
- Testing of software projects to vet security issues
- Continuously track of security properties
- Continuous project security management

Memory and Type Safety

- Two core policies
- Memory safety: safe accesses to memory
- Type Safety: typed accesses to objects

Defense Strategies

- Verify if the complexity of the code is manageable
- Test as much as you can
- Leverage mitigations to constrain the attacker on the remaining attack surface.

Attack Vectors

Goal: understand the goals of an attacker and how these goals may be achieved starting from a program crash.

Case Studies

- Web security (including the browser security model)
- Mobile security

Bonus

- Discussion on shellcode development
- Reverse engineering

Capture-The-Flag!

- Security awareness is an acquired skill. This class heavily involves programming and security exercises.
- A semester long Capture-The-Flag (CTF) to train security skills:
  - Binary analysis
  - Reverse engineering
  - Exploitation techniques
  - Web challenges
- Start: 2019-02-28
- Points are curved: first solver earns more points than last solver; each additional solver reduces points for all previous solvers

Course project (1/2)

- Design and implementation of a project in C++
  - GRASS: GRep AS a Service
  - Allow remote parties to send regular expressions that are then evaluated against a text corpus.
- Security evaluation of your peers' applications
- Fixing any reported security vulnerabilities
- Teams of up to 3 people allowed

Course project (2/2)

- Use a source repository to check in solutions,
- Organize your project according to a design document,
- Peer review and comment the code of other students,
- Work with a large code base, develop extensions.
- C++ primer on Thursday 2019-02-19.

TA Support

- Barooti Khashayar `khashayar.barooti@epfl.ch` "Cryptanalysis of lattice-based post-quantum cryptography."
- Kasra EdalatNejad `kasra.edalat@epfl.ch` "Scaling decentralized privacy-preserving search."
- Solal Pirelli `solal.pirelli@epfl.ch` "Techniques to formally verify real-world software."

Grading

- Lab assignments (CTF): 25% (5 sets of challenges)
- Programming project: 25%
- Midterm: 20% (2019-04-02, 1 hour)
- Final: 30% (2019-05-28, 2 hours)

Academic Integrity

All work that you submit in this course must be your own.
Unauthorized group efforts are considered *academic dishonesty*.
You are allowed to discuss the problem with your peers but you may not copy or reuse any part of an existing solution.
We will use automatic tools to compare your solution to those of other current and past students. The risk of getting caught is too high!

Summary

- Software Security is the area of Computer Science that focuses on (i) testing, (ii) evaluating, (iii) improving, (iv) enforcing, and (v) proving the security of software.
- Learn to identify common security threats, risks, and attack vectors for software systems.
- Assess current security best practices and defense mechanisms for current software systems.
- Design and evaluate secure software.
- Have fun!