CSE 127 Computer Security
Alex Gantman, Spring 2018, Lecture 19

SDLC: Secure Development Lifecycle
Defining Security

- Security is not a functionality feature
  - Most of computer science is about providing functionality:
    - User Interface, Software Design, Algorithms, Operating Systems/Networking, Compilers/PL, Microarchitecture
  - Computer security is not about functionality
    - It is about how the embodiment of functionality behaves in the presence of an adversary.
    - Making sure the system does what it was supposed to do and only what it was supposed to do.

- Holistic property
  - “Software security is about integrating security practices into the way you build software, not integrating security features into your code” – Gary McGraw
Secure Development Lifecycle

▪ How can we develop more robust systems?

▪ Prevention
  – How can we minimize the number of vulnerabilities introduced during development?

▪ Mitigation
  – How can we minimize the impact of vulnerabilities that remain?

▪ Response
  – How can we remediate newly discovered vulnerabilities in deployed products?
Secure Development Lifecycle

Prevention  Mitigation  Response
Secure Development Lifecycle

- Nice story, but... that’s not how it works.
- It all begins with an incident
  - E.g. vulnerability report, exploit release, etc.
Incident Response

- So you get a bug/vulnerability report...
  - Example: Meltdown/Spectre
    - “We have discovered that CPU data cache timing can be abused to efficiently leak information out of mis-speculated execution, leading to (at worst) arbitrary virtual memory read vulnerabilities across local security boundaries in various contexts.”

Incident Response

▪ Step 1: What does it mean?
  – What is the root cause of the vulnerability?
    ▪ Requires deep technical understanding of the system.
    ▪ Have to break through abstraction boundaries.
  – What does it enable an attacker to do?
    ▪ Requires understanding of the threat model (what needs to be protected from whom) and modern exploitation techniques.
  – What is the impact on the user?
    ▪ Requires understanding how products are used by real users.
Incident Response

- Step 2: Address the issue.
  - Develop fix.
    - Ideally, without introducing new bugs in the process.
    - Properly understanding the vulnerability and exploitation techniques is necessary to develop a correct fix.
    - Aside: what if the vulnerability is in third-party code?
  - Propagate fix to impacted product variants.
    - Depending on your industry, you may have many variations of the same software that need to be updated.
    - Have to understand the entire software supply chain and lifecycle.
      - From developer’s keyboard to the landfill.
      - Do you really know what is going into production?
  - Patch/Notify customers.
Incident Response

- Step 3: Analyze the gap.
  - Start working backwards to identify and address gaps in existing process.
  - Response
    - How could you have handled the response better?
  - Mitigation
    - What countermeasures would have mitigated the impact?
  - Prevention
    - How could you have prevented the vulnerability from being released?
Gap Analysis

Prevention → Mitigation → Response
Gap Analysis: Response

- How can you learn about new incidents as early as possible?
  - Was the issue reported privately? Did you learn about it from reading the news?

- Establish cooperative relationships with security researchers
  - Easy to find web page with information on how to report vulnerabilities to the security team
  - Responsive “security@” mail address
  - Encourage/incentivize direct reporting of vulnerabilities
    - If your security program is mature enough, consider a bug bounty

- Aside: Full Disclosure vs Responsible Disclosure vs Coordinated Disclosure
  - If you discover a previously unpublished security vulnerability, I encourage you to report it to the system developers/maintainers
    - Check following directories or try emailing security@<domain>
      - https://hackerone.com/directory
      - https://www.bugcrowd.com/bug-bounty-list/
Gap Analysis

Prevention  Mitigation  Response
Gap Analysis: Mitigation

- What countermeasures would have mitigated the impact?
  - Not all vulnerabilities will be discovered prior to release.

- Countermeasures can make reliable exploitation harder or mitigate the impact of remaining vulnerabilities.
  - Can make exploit development more difficult and costly.
  - Will not stop all exploits.
Gap Analysis: Mitigation

- Identify and implement available countermeasures.
  - Need to understand the current state of the art, cost/benefit tradeoffs, constraints of your platform, etc.
  - Some countermeasures may already be present and only need to be properly configured and enabled.
  - Others may need to be re-implemented or ported on your particular platform.
  - Need continuous process to test proper configuration.

- Research and develop new countermeasures.
  - New offensive and defensive techniques are continuously being developed.
    - Developers introduce new features. Attackers devise ways to exploit these features. Defenders devise new countermeasures. Attackers adapt to the new countermeasures, defenders refine their approach, ...
Gap Analysis

Prevention → Mitigation → Response
Gap Analysis: Prevention

- How could you have prevented the vulnerability from being released?
  - Secure Software Development training for engineers.
  - Automated tools to detect vulnerabilities prior to release.
  - Designing the system with security in mind.
Gap Analysis: Prevention

- Secure Software Training
  - For the last 30 years, most CS graduates had no background in computer security
    - No idea of what types of vulnerabilities are possible, how they are exploited, what they look like, how to fix them, etc.
    - UCSD recently made security a required part of the curriculum.
  - Companies develop custom training to fill the gap
    - Using real in-house code helps make the training relevant.
    - Focus on how to avoid, detect, and address typical software vulnerabilities.
Gap Analysis: Prevention

- Code reviews
- Automated tools to detect vulnerabilities prior to release.
  - Static Analysis
  - Dynamic Analysis
Gap Analysis: Prevention

- Code reviews
  - Pros:
    - A consistently high-yield approach to finding security vulnerabilities
    - Only dependency is access to source
      - No tools to configure, no build system to integrate with, ...
  - Cons:
    - Does not scale well
      - There are a lot of developers writing a lot of code
      - Easy to burn out from continuous code reviews
      - Success highly dependent on expertise of the reviewer
Gap Analysis: Prevention

- To scale, we need to automate

- Static Analysis
  - Compiler-like analysis of the source code, looking for error-prone areas
    - Eg: Coverity, KLOCWork
Unvalidated integer value 'payload' is received from an external function through a call to 'nt32_func' at line 1464 which can be used to access an array through call to 'memcpy' at line 1467.

```c
if (s->msg_callback)
  s->msg_callback(0, a->version, TLS1_SESSION, &s->msg_callback_arg);
if (hastype == TLS1_REQ_PING) {
  unsigned char *tp = s->ssl->rcvbuf[0].pl;
  unsigned short htype;
  unsigned int payload;
  unsigned int padding = 16; /* Use minimum padding */
  /* Read type and payload length first */
  htype = *tp;
  tp += 1;
  if (htype == TLS1_REQ_PING) {
    unsigned char *buffer, *bp;
    int i;
    /* Allocate memory for the response, size is 1 byte
     * message type, plus 2 bytes payload length, plus
     * payload, plus padding */
    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
    bp = buffer;
    /* Enter response type, length and copy payload */
    type = TLS1_RSP_PING;
    xsn_memcpy(bp, pl, payload);
    bp += payload; /* Random padding */
    if (!SND_resend_bytes(bp, padding))
      r = dtls_write_bytes(s->ssl, TLS1_REQ_PING, buffer, 3 + payload + padding);
    if (r > 0 && s->msg_callback)
      s->msg_callback(1, a->version, TLS1_REQ_PING, buffer, 1 + payload + padding, a, s->msg_callback_arg);
    OPENSSL_free(buffer);
    if (r < 0)
      return 0;
  } else if (hastype == TLS1_RSP_PING) {
    ....
  }
  return 0;
```
10367 Cross-site scripting (XSS) injection
10368 Cross-site scripting (XSS) injection
10327 SQL injection
10305 Dereference before null check
10302 Resource leak
10002 Copy-paste error

Show /	argetWebGoat-5.4-\SNAPSHOT\war\main.jsp

```java
if (webSession.getParams() != null) {
    Iterator i = webSession.getParams().iterator();
    while (i.hasNext()) {
        Parameter p = (Parameter) i.next();
        // ...
    }
}
```

10. `taint_path_call` in org.apache.webgoat.session.Parameter.getName() returns the tainted data.
11. `xss_injection_elem` in <html data-tainted="true"> includes `p.getName()` that is tainted data.

**Remediation:** Escaping needs to be done for the detected context. Example:

`Escape.html(taintedData)`

Where `Escape.html` escapes tainted data (for HTML `taintedData` represents the expression `p.getName()`).

- Remove `out.println(printParameters);`
- Replace with `out.println(Escape.html(taintedData));`

13. `xss_sink` in `printParameters` need to be escaped.

Data flow from tainted source to string construction using insufficiently escaped tainted data and then to output:

```
1. into_http_xss_request
2. into_http_xss_response
3. dast_xss_pointentry
4. dast_xss_match

```

Data flow from tainted source to string construction using insufficiently escaped tainted data and then to output:

```
1. tainted_source
2. taint_path_return
3. taint_session
4. taint_path_call
5. taint_path_arg
6. taint_path_param
7. taint_path
8. taint_path_field
9. taint_path_return
10. taint_path_call

```

Occurrence 1 of 2 WebGoat-NTD-DAST
Occurrence 2 of 2 WebGoat-coverty
Gap Analysis: Prevention

- **Static Analysis**
  - **Pros:**
    - Very good at finding some types of vulnerabilities.
      - Custom checkers can be added to detect specific security issues.
  - **Cons:**
    - Can have high false positive rates.
    - No tool works well out of the box.
    - Need to be integrated with the build system and tuned to the idioms of the target codebase.
Gap Analysis: Prevention

- Some vulnerabilities may be easier to discover with dynamic testing
  - Some may only be discoverable with dynamic testing

- Dynamic Analysis/Adversarial Testing/Fuzzing
  - Subject system to _very_ large amounts of tests exercising edge and error cases
  - Combine with instrumentation (sanitizers)
  - Active area of research
  - Example: AFL
    - [http://lcamtuf.coredump.cx/afl/](http://lcamtuf.coredump.cx/afl/)
Fuzzing

- Test case generation
  - Random
  - Mutational
    - Start with a corpus of valid test cases and mutate them
  - Generational
    - Develop a model (grammar) for the input format and use it to generate test cases
Fuzzing

- Code coverage
  - How to measure coverage?
  - Line coverage
  - Branch coverage
  - Path coverage

- Coverage can be a useful feedback function
  - Guide the fuzzer to explore new areas of code

- The real goal though, is to find bugs, not maximize coverage

```c
char bar[20];
if (j > 20)
  return;
for (i = 0; i <= j; i++)
  bar[i] = foo[i];
```
Gap Analysis: Prevention

- **Fuzzing**
  - Pros:
    - Very effective bug-finding technique
    - Highly scalable, can run in the background
  - Cons:
    - Deep bugs are hard to trigger
      - Concolic testing
    - Crash overload
Gap Analysis: Prevention

- Designing the system for security and containment of compromise.
  - Defining a threat model (what should be protected from whom)
  - Identifying and evaluating risks
  - Building in defenses
Risk Assessment

- Risk assessment/risk analysis process for assessing risk in a system:
  1. Start by understanding system requirements
  2. Identify assets and attackers
  3. Establish security requirements
  4. Evaluate system design
  5. Identify threats and classify risks
  6. Address identified risks
Gap Analysis: Prevention

- Building in defenses
  - Isolation of components to contain damage from spill-over
  - Hardening of security boundaries
  - Re-architecting of high-risk components for sandboxing
Secure Development Lifecycle

- Prevention
- Mitigation
- Response
Secure Development Lifecycle

- A methodology for building more secure systems
- Many variations on the same theme
  - Microsoft SDLC
  - Cigital Touchpoints
  - Building Security In Maturity Model (BSIMM)
  - SafeCode
  - ...

Secure Development Lifecycle

- Microsoft SDLC
  - 1990’s and Code Red
  - Bill Gates’ Trustworthy Computing Memo
Code Red

- "On July 19, 2001 more than 359,000 computers were infected with the Code-Red (CRv2) worm in less than 14 hours. At the peak of the infection frenzy, more than 2,000 new hosts were infected each minute."
  - [https://www.caida.org/research/security/code-red/coderedv2_analysis.xml](https://www.caida.org/research/security/code-red/coderedv2_analysis.xml)
Bill Gates’ Trustworthy Computing Memo

  - January 2002
  - “So now, when we face a choice between adding features and resolving security issues, we need to choose security. Our products should emphasize security right out of the box, and we must constantly refine and improve that security as threats evolve... If we discover a risk that a feature could compromise someone's privacy, that problem gets solved first. If there is any way we can better protect important data and minimize downtime, we should focus on this. These principles should apply at every stage of the development cycle of every kind of software we create, from operating systems and desktop applications to global Web services.”
Secure Development Lifecycle

- Microsoft SDLC

<table>
<thead>
<tr>
<th>1. TRAINING</th>
<th>2. REQUIREMENTS</th>
<th>3. DESIGN</th>
<th>4. IMPLEMENTATION</th>
<th>5. VERIFICATION</th>
<th>6. RELEASE</th>
<th>7. RESPONSE</th>
</tr>
</thead>
</table>
Secure Development Lifecycle

Microsoft products: Vulnerabilities reduction after SDL implementation

Windows:
- 45% reduction of vulnerabilities disclosed one year after release
- Before SDL: 119
- After SDL: 66

SQL Server:
- 91% reduction of vulnerabilities disclosed three years after release
- Before SDL 2000: 34
- After SDL 2000: 3
- Before SDL 2005: 3
- After SDL 2005: 3

Sources: Microsoft Security Blog and Microsoft TechNet Security Blog
Secure Development Lifecycle

- Cigital Touchpoints
  - [http://www.swsec.com/resources/touchpoints/](http://www.swsec.com/resources/touchpoints/)
Secure Development Lifecycle

- **BSIMM**
  - [https://www.bsimm.com/](https://www.bsimm.com/)
  - “*multiyear study of [109] real-world software security initiatives*”
  - 113 activities across 12 practices, organized into four domains
Secure Development Lifecycle

- Caveat emptor
  - The real development process is much more complex
  - Details will depend on the technology of your product, the nature of your business, culture of your company, ...
Security projects

Requested feature  Documentation  Analyzed design  Proposed mitigations  Reviewed implementation

Pentested configuration  Deployment  BlackHat  Version 2.0 (we take security seriously)  Competitor's product

Create your own cartoon at www.projectcartoon.com
Ethics

- In this class you learned how to attack the security of computer systems (and some physical systems)
- We learn attacks because it is needed to understand how to defend them
- You have an obligation to use this knowledge ethically
  - You may not attack others
  - Many good legitimate hacking challenges.
Exploit Playgrounds

- Wargames
  - http://overthewire.org/wargames/

- Reverse Engineering Challenges
  - https://challenges.re/

- CTFs
  - https://ctftime.org/ctfs
  - UCSD CTF: https://cseweb.ucsd.edu/~dkohlbre/ctf/index.html
Course Objectives

- A solid foundation of security concepts, backed by concrete examples

- Security mindset
  - How to think like an attacker/security engineer
  - Looking beyond the system’s intended functionality, to what it can be made to do

- Understanding how things work, how they break, and how to fix them
  - Technical details of vulnerabilities, attacks, and defenses
Course Objectives

During your career you will design and build complex systems. With probability asymptotically approaching 1, you will introduce numerous security vulnerabilities in the process.

My goals are to help you:

a) Minimize the number and severity of vulnerabilities you will introduce;

b) Better understand the root causes and impact of vulnerabilities that are brought to your attention;

c) Properly address identified vulnerabilities.
Prerequisites/Expectations

- You are expected to have a basic understanding of
  - C and assembly
  - Operating Systems
  - Computer Architecture
  - Networking
Homework

- Final is on Saturday 6/9 3pm-6pm in WLH 2005
  - Covers everything: from first lecture up to, and including, today
  - Similar format to the midterm
  - You can bring in one sheet (8.5”x11”) of paper with notes
Thank You