Welcome to

8. Web Application Security: Defensive. Code review, automation, patterns/anti-patterns

Security in Web Development Elective, KEA

Henrik Kramselund he/him han/ham hkj@zencurity.com @kramse 🐦

Slides are available as PDF, kramse@Github

8-defensive-automation-security-in-web.tex in the repo security-courses

Goals for today

Todays goals:

- Doing defensive
- Try a few tool runs, PMD, Git hooks and AFL

Photo by Thomas Galler on Unsplash

Plan for today

Subjects

- Process for starting code review
- A few patterns and anti-patterns

Exercises

- PMD static code Analysis
- Git hooks for preventing source entering repository
- Look at a fuzzer American fuzzy lop

Time schedule

- 1) 19. Reviewing Code for Security 45min
- 2) 20. Vulnerability Discovery 45min
- 3) Running various tools, discussing tools 45min
- 4) Fuzzing – try AFL fuzzer 45min

We will not follow this to the minute, but be more flexible today.

Reading Summary

Web Application Security, Andrew Hoffman, 2020, ISBN: 9781492053118

Part III. Defense, chapters 19-21

19. Reviewing Code for Security

20. Vulnerability Discovery

21. Vulnerability Management

Use the secure coding PDF from Veracode

Chapter 21. contains information you should know, when reading about CVSS scores later - but we will not go through this chapter in detail

19. Reviewing Code for Security - introduction

The code review stage must always occur after the architecture stage in a security-conscious organization, and never before.

Some technology companies today uphold a "move fast and break things" mantra, but such a philosophy often is abused and used as a method of ignoring proper security processes. Even in a fast-moving company, it is imperative that application architecture is reviewed prior to shipping code. Although from a security perspective it would be ideal to review the entire feature architecture upfront, this may not be feasible in uncertain conditions. As such, **at a minimum the major and well-known features should be architected and reviewed, and when new features come up they should be both architected and reviewed for security prior to development as well**.

The proper time to review code for security gaps is once the architecture behind the code commit has been properly reviewed. This means code reviews should be the second step in an organization that follows secure development best practices.

- Note: the mention of new features, architect, then review!

How to Start a Code Review

A code security review should operate very similarly to a code functionality review.

1. Check out master with git checkout master .
2. Fetch and merge the latest master with git pull origin master .
3. Check out the feature branch with git checkout <username>/feature .
4. Run a diff against the master with git diff origin/master...

The git diff command should return two things:

- A list of files that differ on master and the current branch
- A list of changes in those files between master and the current branch

Being able to use Git is important for most programmers today!

What to look for

Archetypical Vulnerabilities Versus Custom Logic Bugs

A code functionality review checks code to ensure it meets a feature spec and does not contain usability bugs. A code security review checks for common vulnerabilities such as XSS, CSRF, injection, and so on, but more importantly checks for logic-level vulnerabilities that require deep context into the purpose of the code and cannot be easily found by automated tools or scanners.

In order to find vulnerabilities that arise from logic bugs, we need to first have context in regard to the goal of the feature. This means we need to understand the users of the feature, the functionality of the feature, and the business impact of the feature.

- We will mostly cover generic bugs, and a talk about process and tools today

Where to look, first

To summarize, an effective way of determining what code to review in a security review of a web application is as follows:

1. Evaluate the client-side code to gain understanding of the business logic and understand what functionality users will be capable of using.
2. Using knowledge gained from the client review, begin evaluating the API layer, in particular, the APIs you found via the client review. In doing this, you should be able to get a good understanding of what dependencies the API layer relies on to function.
3. Trace the dependencies in the API layer, carefully reviewing databases, helper libraries, logging functions, etc. In doing this, you will get close to having covered the majority of user-facing functionality.
4. Using the knowledge of the structure of the client-linked APIs, attempt to find any public-facing APIs that may be unintentionally exposed or intended for future feature releases. Review these as you find them.
5. Continue on throughout the remainder of the codebase. This should actually be pretty easy because you will already be familiar with the codebase having read through it in an organic method versus trying to brute force an understanding of the application architecture.

Secure-Coding Anti-Patterns: Block list

In the world of security, mitigations that are temporary should often be ignored and instead a permanent solution should be found, even if it takes longer.

```javascript
const whitelist = ['https://happy-site.com', 'https://www.my-friends.com'];
/*
* Determine if the domain is allowed for integration.
*/
const isDomainAccepted = function(domain) {
return whitelist.includes(domain);
};
```

- Block lists are temporary
- Allow lists are more secure, what to allow

Boilerplate Code

Another security anti-pattern to look for is the use of boilerplate or default framework code. This is a big one, and easy to miss because often frameworks and libraries require effort to tighten security, when they really should come with heightened secu- rity right out of the box and require loosening.

- Beware when using default examples from books, libraries etc.
- Use a secure by default approach instead
- Use firewalls always, and only open access to services when needed

Chapter 20. Vulnerability Discovery

After securely architected code has been designed, written, and reviewed, a pipeline should be put in place to ensure that no vulnerabilities slip through the cracks.

Typically, applications with the best architecture experience the least amount of vulnerabilities and the lowest risk vulnerabilities. After that, applications with sufficiently secure code review processes in place experience fewer vulnerabilities than those without such processes, but more than those with a secure-by-default architecture.

- Security Automation
- Static Analysis

```
main(int argc, char **argv)
{         char buf[200];
          strcpy(buf, argv[1]);
          printf("%s\n",buf);

}
```

Use tools for analysing code and applications

Static Analysis

Run through the programsource code or binary, without running it

Can find bad functions like strcpy

Dynamic Analysis

Running the program with a test-harness, monitoring system calls, memory operations etc.

Static tools

Flawfinder http://www.dwheeler.com/flawfinder/

RATS Rough Auditing Tool for Security, C, C++, Perl, PHP and Python

PMD static ruleset based (Java—free)

Checkmarx (most major languages—paid)

Bandit (Python—free)

Brakeman (Ruby—free)

ESLint JavaScript https://eslint.org/

http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

label

PMD An extensible cross-language static code analyzer.

About PMD

PMD is a source code analyzer. It finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth. It supports Java, JavaScript, Salesforce.com Apex and Visualforce, PLSQL, Apache Velocity, XML, XSL.

Additionally it includes CPD, the copy-paste-detector. CPD finds duplicated code in Java, C, C++, C#, Groovy, PHP, Ruby, Fortran, JavaScript, PLSQL, Apache Velocity, Scala, Objective C, Matlab, Python, Go, Swift and Salesforce.com Apex and Visualforce.

https://pmd.github.io/

Exercise

Now lets do the exercise

Trying PMD static analysis 30 min

which is number **39** in the exercise PDF.

A Fool with a Tool is still a Fool

1. Run tool
2. Fix problems
3. Rinse repeat

Fixing problems?

```
char tmp[256]; /* Flawfinder: ignore */
strcpy(tmp, pScreenSize); /* Flawfinder: ignore */
```

Example from http://www.dwheeler.com/flawfinder/

Coding standards - style

This file specifies the preferred style for kernel source files in the OpenBSD source tree. It is also a guide for preferred user land code style. These guidelines should be followed for all new code. In general, code can be considered "new code" when it makes up about 50more of the file(s) involved. ...

Use queue(3) macros rather than rolling your own lists, whenever possible. Thus, the previous example would be better written:

```c
#include <sys/queue.h>
struct  foo {
LIST_ENTRY(foo) link;   /* Queue macro glue for foo lists */
        struct  mumble amumble; /* Comment for mumble */
        int     bar;
};
LIST_HEAD(, foo) foohead;    /* Head of global foo list */
```

OpenBSD style(9)

Coding standards functions

The following copies as many characters from input to buf as will fit and NUL terminates the result. Because strncpy() does not guarantee to NUL terminate the string itself, it must be done by hand.

```
char buf[BUFSIZ];

(void)strncpy(buf, input, sizeof(buf) - 1);
buf[sizeof(buf) - 1] = '\0';
```

Note that strlcpy(3) is a better choice for this kind of operation. The equivalent using strlcpy(3) is simply:

```
(void)strlcpy(buf, input, sizeof(buf));
```

OpenBSD strcpy(9)

Compiler warnings - gcc -Wall

```
# gcc -o demo demo.c
demo.c: In function main:
demo.c:4: warning: incompatible implicit declaration of built-in
function strcpy


# gcc -Wall -o demo demo.c
demo.c:2: warning: return type defaults to int
demo.c: In function main:
demo.c:4: warning: implicit declaration of function strcpy
demo.c:4: warning: incompatible implicit declaration of built-in
function strcpy
demo.c:5: warning: control reaches end of non-void function
```

Easy to do!

No warnings = no errors?

```
# cat demo2.c
#include <strings.h>
int main(int argc, char **argv)
{
    char buf[200];
    strcpy(buf, argv[1]);
    return 0;
}
# gcc -Wall -o demo2 demo2.c
```

This is an insecure program, but no warnings!

(cheating, some compilers actually warn today)

Version control sample hooks scripts

Before checking in code in version control, pre-commit - check

- case-insensitive.py
- check-mime-type.pl
- commit-access-control.pl
- commit-block-joke.py
- detect-merge-conflicts.sh
- enforcer
- log-police.py
- pre-commit-check.py
- verify-po.py

http://subversion.tigris.org/tools_contrib.html

http://svn.collab.net/repos/svn/trunk/contrib/hook-scripts/

This references Subversion, which is not used much anymore. Just to show the concept is NOT new. Use hooks!

Example Enforcer

In a Java project I work on, we use log4j extensively. Use of
System.out.println() bypasses the control that we get from log4j,
so we would like to discourage the addition of println calls in
our code.

We want to deny any commits that add a println into the code.
The world being full of exceptions, we do need a way to allow
some uses of println, so we will allow it if the line of code
that calls println ends in a comment that says it is ok:

```
System.out.println("No log4j here"); // (authorized)
```

http://svn.collab.net/repos/svn/trunk/contrib/hook-scripts/enforcer/enforcer

Example verify-po.py

```python
#!/usr/bin/env python
"""This is a pre-commit hook that checks whether the contents
of PO files committed to the repository are encoded in UTF-8.
"""
```

http://svn.collab.net/repos/svn/trunk/tools/hook-scripts/verify-po.py

Design for security - more work
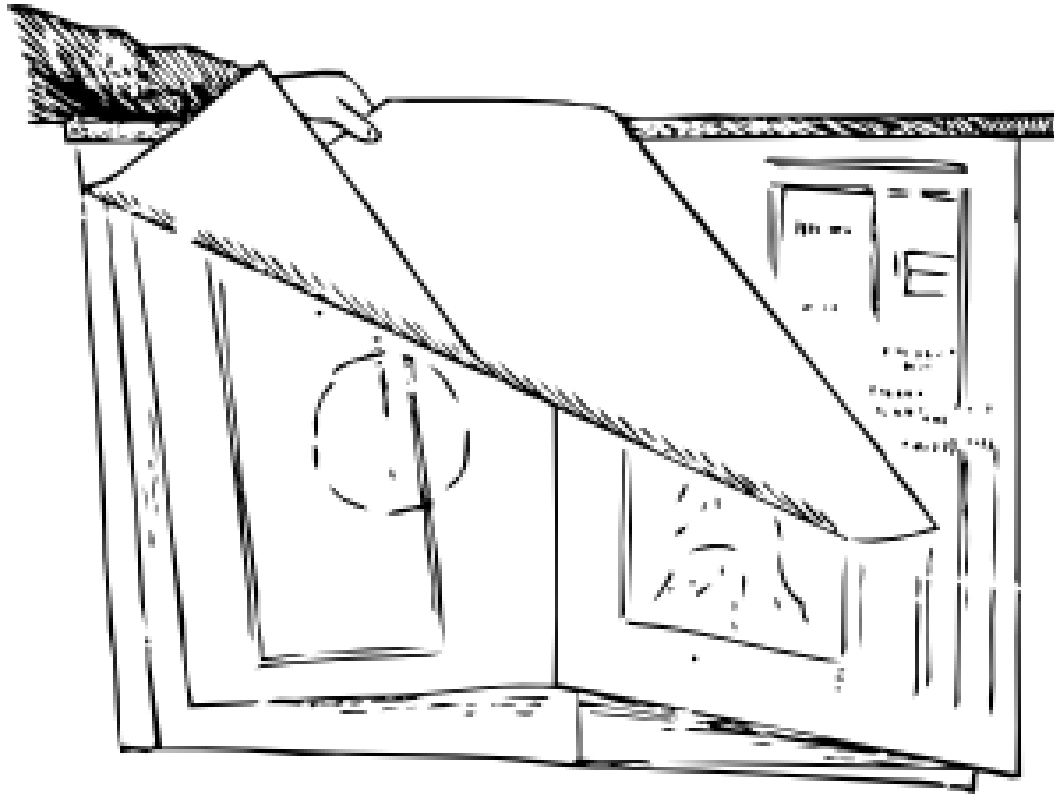
Security is cheapest and most effective when done during design phase.

Unit testing - low level / functions

```java
public class TestAdder {
    public void testSum() {
        Adder adder = new AdderImpl();
        assert(adder.add(1, 1) == 2);
        assert(adder.add(1, 2) == 3);
        assert(adder.add(2, 2) == 4);
        assert(adder.add(0, 0) == 0);
        assert(adder.add(-1, -2) == -3);
        assert(adder.add(-1, 1) == 0);
        assert(adder.add(1234, 988) == 2222);
    }
}
```

Test individual functions

Example from http://en.wikipedia.org/wiki/Unit_testing

Avoid regressions, old errors reappearing

Book references Jest testing library

The difference between a functional test and a vulnerability test is not the framework but the purpose for which the test was written.

...

Vulnerability regression tests are simple. Sometimes they are so simple, they can be written prior to a vulnerability being introduced. This can be useful for code where minor insignificant-looking changes could have a big impact. Ultimately, vulnerability regression testing is a simple and effective way of preventing vulnerabilities that have already been closed from reentering your codebase.

The tests themselves should be run on commit or push hooks when possible (reject the commit or push if the tests fail). Regularly scheduled runs (daily) are the second-best choice for more complex version control environments.

Exercise

Now lets do the exercise

Git hook 30 min

which is number **40** in the exercise PDF.

Break it

Use fuzzers, hackertools, improve security by breaking it

Fuzzing

Fuzz testing or Fuzzing is a Black Box software testing technique, which basically consists in finding implementation bugs using malformed/semi-malformed data injection in an automated fashion.
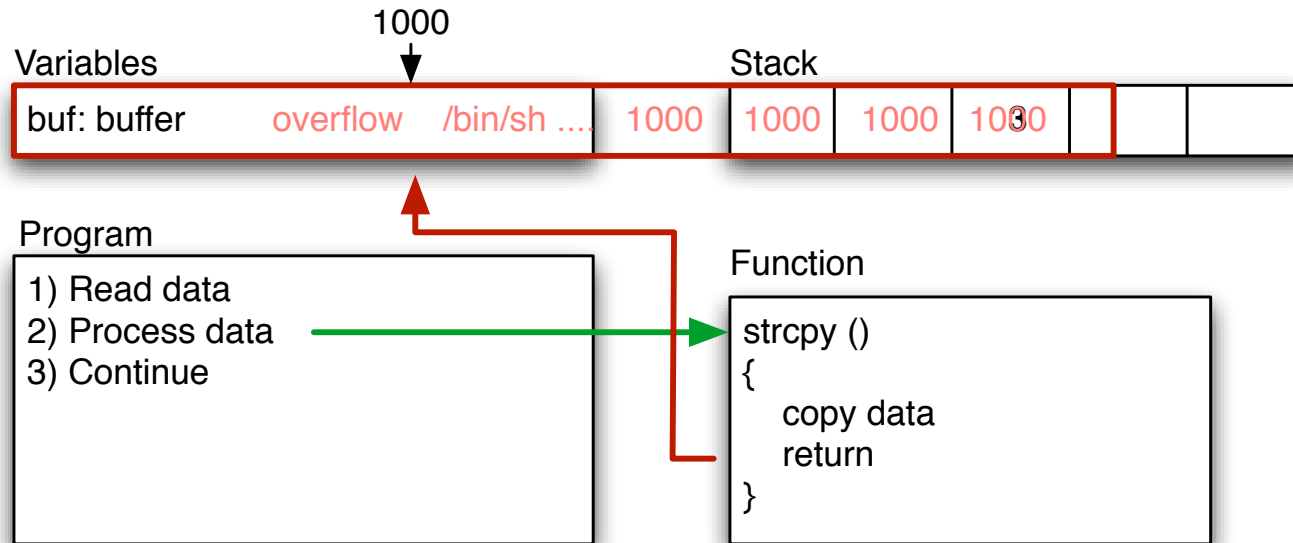
A trivial example Let's consider an integer in a program, which stores the result of a user's choice between 3 questions. When the user picks one, the choice will be 0, 1 or 2. Which makes three practical cases. But what if we transmit 3, or 255 ? We can, because integers are stored a static size variable. If the default switch case hasn't been implemented securely, the program may crash and lead to "classical" security issues: (un)exploitable buffer overflows, DoS, ...

History Fuzz testing was developed at the University of Wisconsin Madison in 1989 by Professor Barton Miller and students. Their (continued) work can be found at http://www.cs.wisc.edu/ bart/fuzz/ ; it's mainly oriented towards command-line and UI fuzzing, and shows that modern operating systems are vulnerable to even simple fuzzing.

Sources: https://owasp.org/www-community/Fuzzing
https://owasp.org/www-project-web-security-testing-guide/v41/6-Appendix/C-Fuzz_Vectors

Overflow – segmentation fault

1000

Variables

Stack

| buf: buffer | overflow | /bin/sh .... | 1000 | 1000 | 1000 | 1000 | | | |

Program

1) Read data
2) Process data
3) Continue

Function

```
strcpy ()
{
    copy data
    return
}
```

- Bad function overwrites return value!
- Control return address
- Run shellcode from buffer, or from other place

Simple fuzzer

```
$ for i in 10 20 30 40 50
>> do
>> ./demo `perl -e "print 'A'x$i"`
>> done
AAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Memory fault
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Memory fault
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Memory fault
```

Memory fault/segmentation fault - juicy!

Note about web fuzzing vs command line

While the end goal may be to find a vulnerability that can be exploited through a web application, it is much more efficient to fuzz smaller parts independently.

So better to:

- Exploit a program while being able to monitor crashes
- Run a function with all inputs, without having to HTML encode everything first
- Run specific functions
- Run multiple instances of fuzzer in parallel directly – faster

See the crash remotely is hard, if not impossible

Exercise
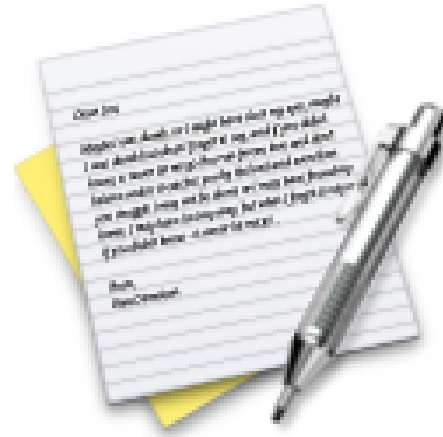
Now lets do the exercise

Try American fuzzy lop up to 45min

which is number **41** in the exercise PDF.
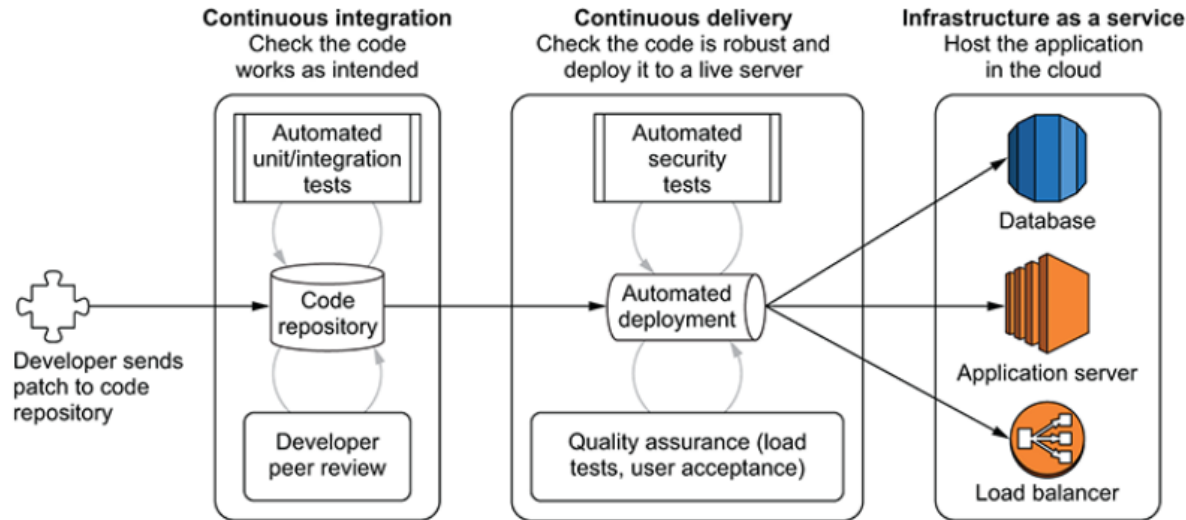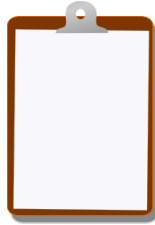
Putting it all together

Figure 1.2 Continuous integration (CI), continuous delivery (CD), and infrastructure as a service (IaaS) form an automated pipeline that allows DevOps to speed up the process of testing and deploying software.

Source: *Securing DevOps: Security in the cloud* by Julien Vehent Manning 2018, 384 pages

For Next Time

Think about the subjects from this time, write down questions

Check the plan for chapters to read in the books

Visit web sites and download papers if needed

Retry the exercises to get more confident using the tools