Fakultät für Informatik und Mathematik

6090: Security of Computer and Embedded Systems

Week 5: Security Testing; Security of Third-party Components

Elif Bilge Kavun

elif.kavun@uni-passau.de

November 16, 2021

This Week’s Outline

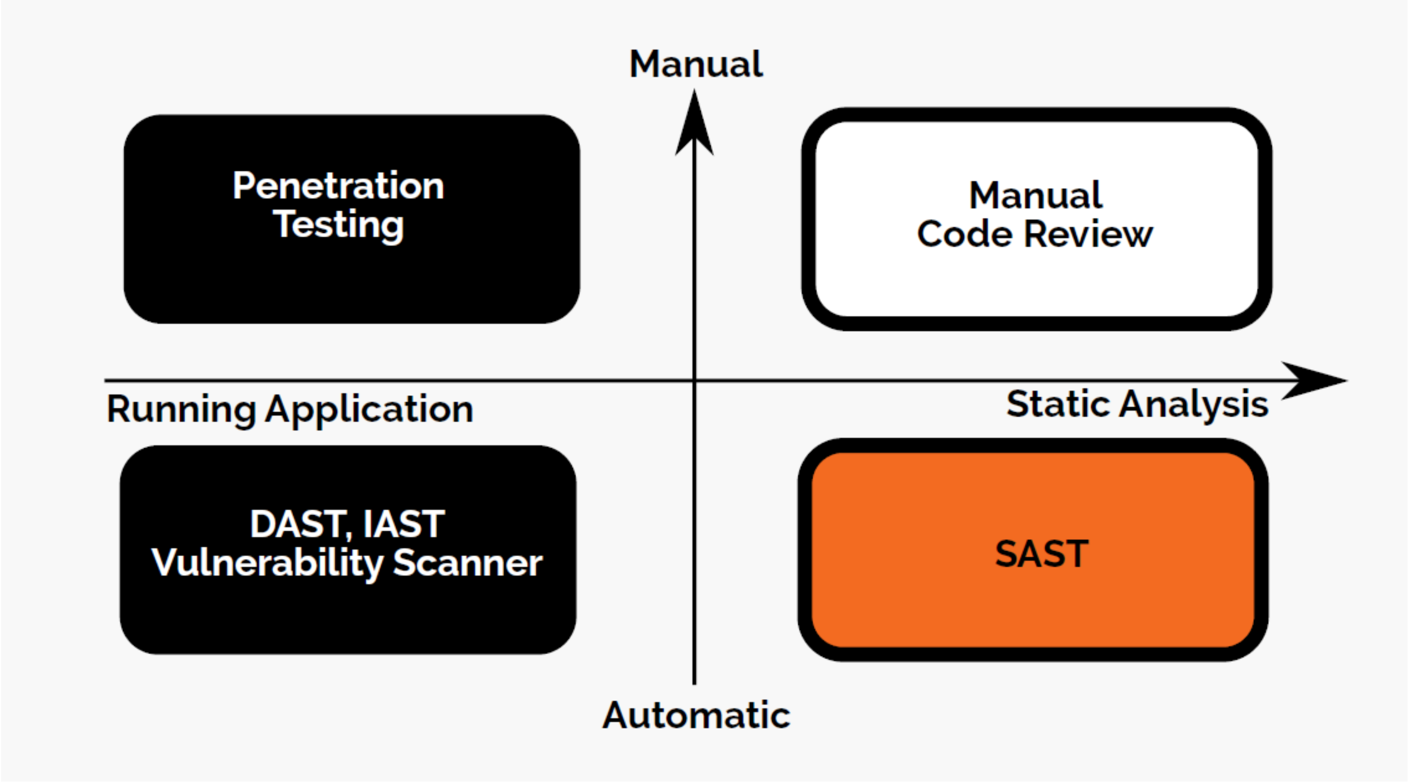
- Security Testing
  - Overview
  - Static/Dynamic Analysis
  - Fuzzing
- Security of Third-party Components

Security Testing

Overview & Static/Dynamic Analysis

Static Analysis Overview

- Finding security vulnerabilities
  - Static application security testing (SAST)



Static Analysis Overview

- Is everything secure?

“Our tool reports all vulnerabilities in your software – you only need to fix them and you are secure.”

Undisclosed sales engineer from a SAST tool vendor

  - This tool is called “Code Assurance Tool (cat)”
  - The cat tool reports each line that might contain a vulnerability
  - It supports also a mode that reports no ***false positives***

Static Analysis Overview

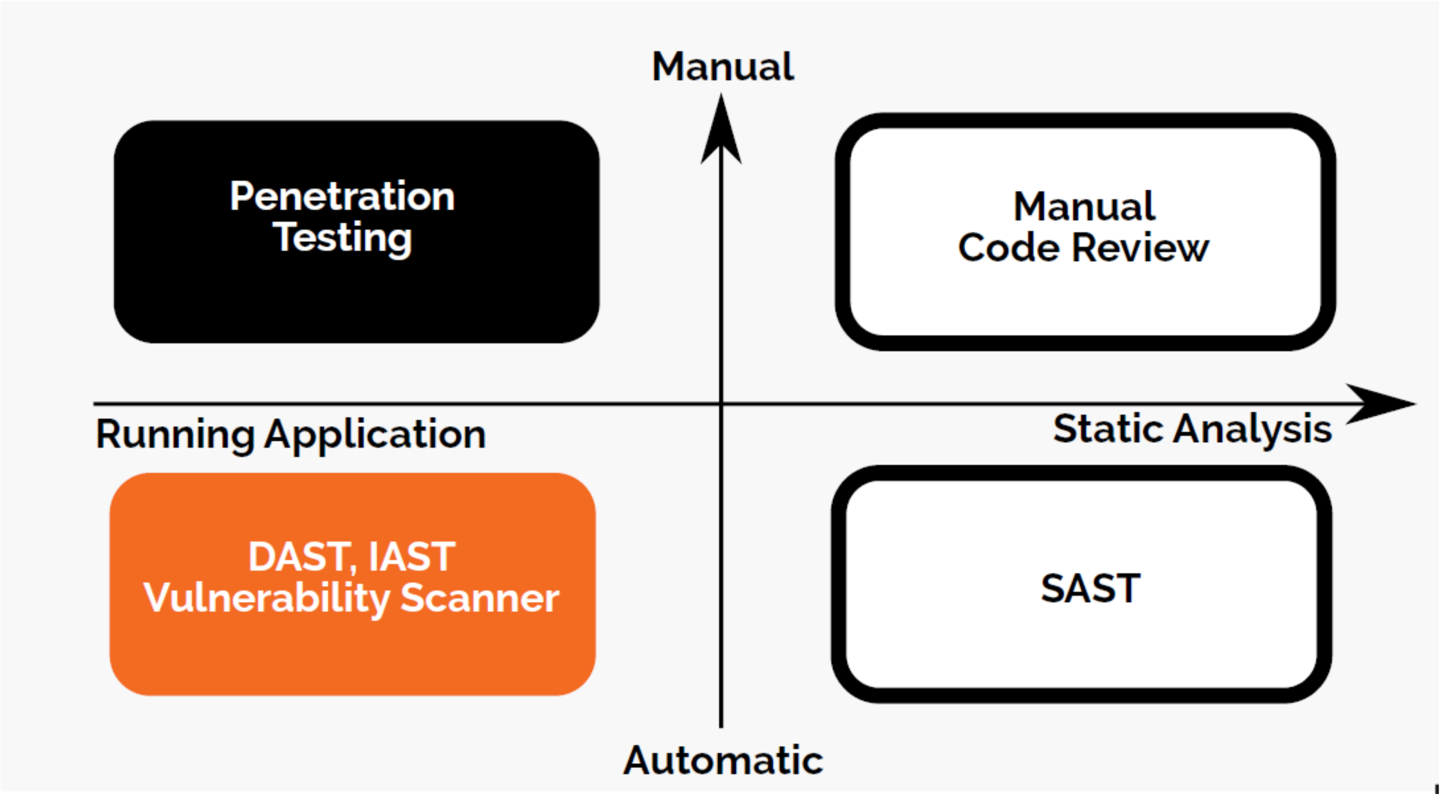
- What We Want to Find: Programming Patterns That May Cause Security Vulnerabilities

- Mainly two patterns
  - Local issues (no data-flow dependency), e.g.,
    - Insecure functions
    - Secrets stored in the source code
  - Data-flow related issues, e.g.,
    - Cross-site Scripting (XSS)
    - Secrets stored in the source code

- Trust own developers, i.e., focus on finding "obvious" bugs

Dynamic Analysis Overview

- Finding security vulnerabilities
  - Dynamic application security testing (DAST)



Dynamic Analysis Overview

- Sniffers and Proxies
  - Tools for inspecting network traffic

- Sniffers
  - Observe traffic in real-time
  - Capture traffic for later analysis (or replay)
  - No modification of traffic/systems
  - Examples: Wireshark, tcpdump

- Intercepting Proxies
  - Observe and capture traffic
  - Can block traffic
  - Can change traffic/content "in transit"
  - Modify traffic of systems
  - Examples: mitmproxy, Fiddler, OWASP ZAP

Dynamic Analysis Overview

- Sniffers and Proxies
  - Tools for inspecting network traffic

- Sniffers
  - Observe traffic in real-time
  - Capture traffic for later analysis (or replay)
  - No modification of traffic/systems
  - Examples: Wireshark, tcpdump

SAFE

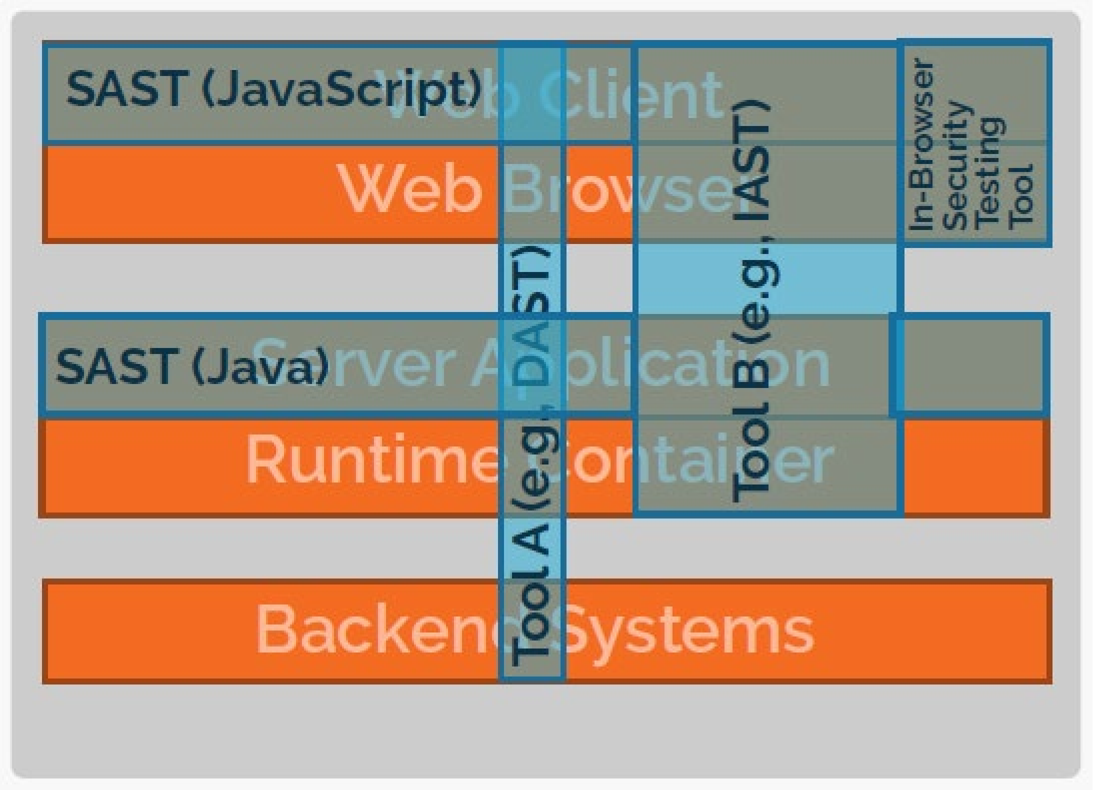
- Intercepting Proxies
  - Observe and capture traffic
  - Can block traffic
  - Can change traffic/content "in transit"
  - Modify traffic of systems
  - Examples: mitmproxy, Fiddler, OWASP ZAP

DANGEROUS

Combining Static and Dynamic Security Testing

- Combining Multiple Security Testing Methods and Tools: Different Test Focus

- Risks of only using only SAST
  - Wasting effort that could be used more wisely elsewhere
  - Shipping insecure software
- Examples of SAST limitations
  - Not all programming languages supported
  - Covers not all layers of the software stack
- A comprehensive approach combines
  - Static approaches (i.e., SAST)
  - Dynamic approaches (i.e., IAST or DAST)



Combining Static and Dynamic Security Testing

- Combining Multiple Security Testing Methods and Tools: Different Risk
  - Using static analysis has a low security risk
    - In the worst case, you will learn potential vulnerabilities in your software
  - Dynamic security testing is dangerous!
    - You might
      - Break your productive IT landscape (e.g., mistyping an IP address)
      - Destroy/corrupt your database (e.g., testing SQL injections)
      - Violate compliance policies (granting access to data you should not see)
  - These are no reasons not to use dynamic tests
    - Approval from the IT department might be necessary
    - Dedicated test systems (and infrastructure)
    - Necessary expertise (security and domain/app knowledge)

False Positives and False Negatives

- Are all results real issues?
  - Both static and dynamic tools suffer from false positives (and false negatives)

	Weakness Exists	No Weakness Exists
Weakness Reported	True Positive	False Positive
No Weakness Reported	False Negative	True Negative

„Positive“ = A weakness **reported**

„Negative“ = **No** weakness **reported**

„True“ ::= (reported *and* exists)

or

(not-reported *and* not-exists)

False Positives and False Negatives

- An informal definition:
  - If a static analysis tool reports a finding
    - It can be exploitable (true positive)
    - It cannot be exploitable (false positive)
  - If a static analysis tool does not report a finding
    - The code is secure (true negative)
    - The code contains a vulnerability (false negative)

False Positives and False Negatives

- Let us take the view point of a
  - Developer
  - Security expert

False Positives and False Negatives

- Let us take the view point of a
  - Developer: "I want a tool with zero false positives!"
  - Security expert

False Positives and False Negatives

- Let us take the view point of a
  - Developer: "I want a tool with zero false positives!"
  - Security expert: "I want a tool with zero false negatives!"

False Positives and False Negatives

- Let us take the view point of a
    - Developer: "I want a tool with zero false positives!"
        - False positives create unnecessary effort
    - Security expert: "I want a tool with zero false negatives!"

False Positives and False Negatives

- Let us take the view point of a
  - Developer: "I want a tool with zero false positives!"
    - False positives create unnecessary effort
  - Security expert: "I want a tool with zero false negatives!"
    - False negatives increase the overall security risk

False Negatives
Reasons and Recommendations (Examples)

- Fundamental: Under-approximation of the tool (method), e.g.,
    - Missing language features (might intercept data flow analysis)
    - Missing support for complete syntax (parsing errors)
  - Report to tool vendor!

- Configuration: Lacking knowledge of insecure frameworks, e.g.,
    - Insecure sinks (output) and sources (input)
  - Improve configuration!

- Unknown security threats: For us, e.g.,
    - XML verb tampering
  - Develop new analysis for tool! (Might require support from tool vendor)

False Positives
Reasons and Recommendations (Examples)

- Fundamental: Over-approximation of the tool (method), e.g.,
    - Pointer analysis
    - Call stack
    - Control-flow analysis
  - Report to tool vendor!

- Configuration: Lacking knowledge of security framework, e.g.,
    - Sanitization functions
    - Secure APIs
  - Improve configuration!

- Mitigated by attack surface: Strictly speaking a true finding, e.g,
    - No external communication due to firewall
    - SQL injections in a database admin tool
  - Should be fixed!
  - In practice often mitigated during audit, or local analysis configuration

Prioritization of Findings
A Pragmatic Solution for Too Many Findings

- What needs to be audited?
- What needs to be fixed?
  - As security issue (response effort)
  - Quality issue
- Different rules for
  - Old code
  - New code

Security Testing

Fuzzing

Fuzzing
The Origins

- The term "fuzzing" originates from a 1988 class project, taught by Barton Miller at the University of Wisconsin
- Core idea
  - Create large random strings
  - Pipe input into UNIX utilities
  - Check if they crash

```
$ fuzz 100000 -o outfile | deqn
```

- Very simple, but very effective


Barton P. Miller, Lars Fredriksen and Bryan So

An Empirical Study of the Reliability of UNIX Utilities

Fuzzing
Challenges

- Detecting input channel
  - Tested UNIX utilities accept input as command line argument or via STDIN
  - In contrast, modern scenarios require support for various protocols and data types (e.g., WebRequests using JSON)
- Input generation
  - Tested UNIX tools accepted any string as input
  - In contrast, modern scenarios often require valid input files (e.g. SQL, JPG, JavaScript)
- Deciding if the response is a bug (vulnerability) or not
  - Tested Unix utilities crashed ("core dump") or hung
  - In contrast, modern scenarios are more complex, e.g., fuzzing for finding SQL injections
    - How does the correct response look like?
    - How does an "exploit" look like?
    - What is the assessment for a data base error?
- When did we fuzz enough (coverage)?

Fuzzing
Random Fuzzing

- Very simple

- Inefficient
  - Random input is often reject, as a specific format is required
  - Probability of causing a crash is very low

- Likely to generate random HTML documents
  - <html></html>
  - <html>AAAA</html>
  - <html></html></html>
  - <html>/</<>></html>
  - <html></body>&</body></html>

Fuzzing
Mutation-based Fuzzing

- Idea: Mutate existing data samples to create test data
- Little or no knowledge of the structure of the inputs is assumed
- Anomalies are added to existing valid inputs
- Anomalies may be completely random or follow some heuristics
- Requires little to no set up time
- Dependent on the inputs being modified
- May fail for protocols with checksums, those which depend on challenge response, etc.
- Example Tools: Taof, GPF, ProxyFuzz, Peach Fuzzer

Fuzzing
Mutation-based Fuzzing

- Advantages
  - Easy to setup and automate
  - Requires little to no knowledge of the input format/protocol

- Disadvantages
  - Effectiveness limited by selection of initial data set
  - Has problems with file formats/protocols that require valid checksums

Fuzzing
Generation-based Fuzzing

- Idea: Define new tests based on models (specifications) of the input format
- Generate random inputs with the input specification in mind (RFC, documentation, etc.)
- Add anomalies to each possible spot
- Knowledge of the input format allows to prune input that are rejected by the application
- Input can be specified by a grammar (grammar-based fuzzing)
- Example tools: SPIKE, Sulley, Mu-4000, Peach Fuzzer

Fuzzing
Generation-based Fuzzing

- Advantages
  - Completeness (you can measure how much of the specification has been covered)
  - Can handle complex inputs (e.g., that require matching checksums)

- Disadvantages
  - Building a generator can be a complex problem
  - Specification needs to be available

Fuzzing
Advanced Fuzzing Techniques

- Idea: Generate input based on behavior/responses of the program
  - Greybox-Fuzzing (Concolic Testing)
    - Uses symbolic execution to trigger unused paths
    - Invented by Microsoft and used for fuzzing file input routines (e.g., in MS Office)
  - Autodafe
    - Fuzzing by weighting attacks with markers
    - Open Source
  - Evolutionary Fuzzing System (EFS)
    - Generate tests cases/inputs based on code coverage metrics
  - AFL (American Fuzzy Loop) and LibFuzzer
    - Compile time instrumentation (coverage) and genetic algorithms (mutations)

Security of Third-party Components

How We Develop Software

- How it used to be

```c
#include <stdio.h>

int main (void) {
  printf ("Hello, world!\n");
  return 0;
}
```

"hello.c"

- Only few external dependencies

  ("HelloWorld" only requires system libs)
- Full control over source code

- How we do it today

```javascript
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello, World!');
});

app.listen(3000, function () {
  console.log('Hello app listening on port 3000!');
});
```

"hello.js"

- Many dependencies

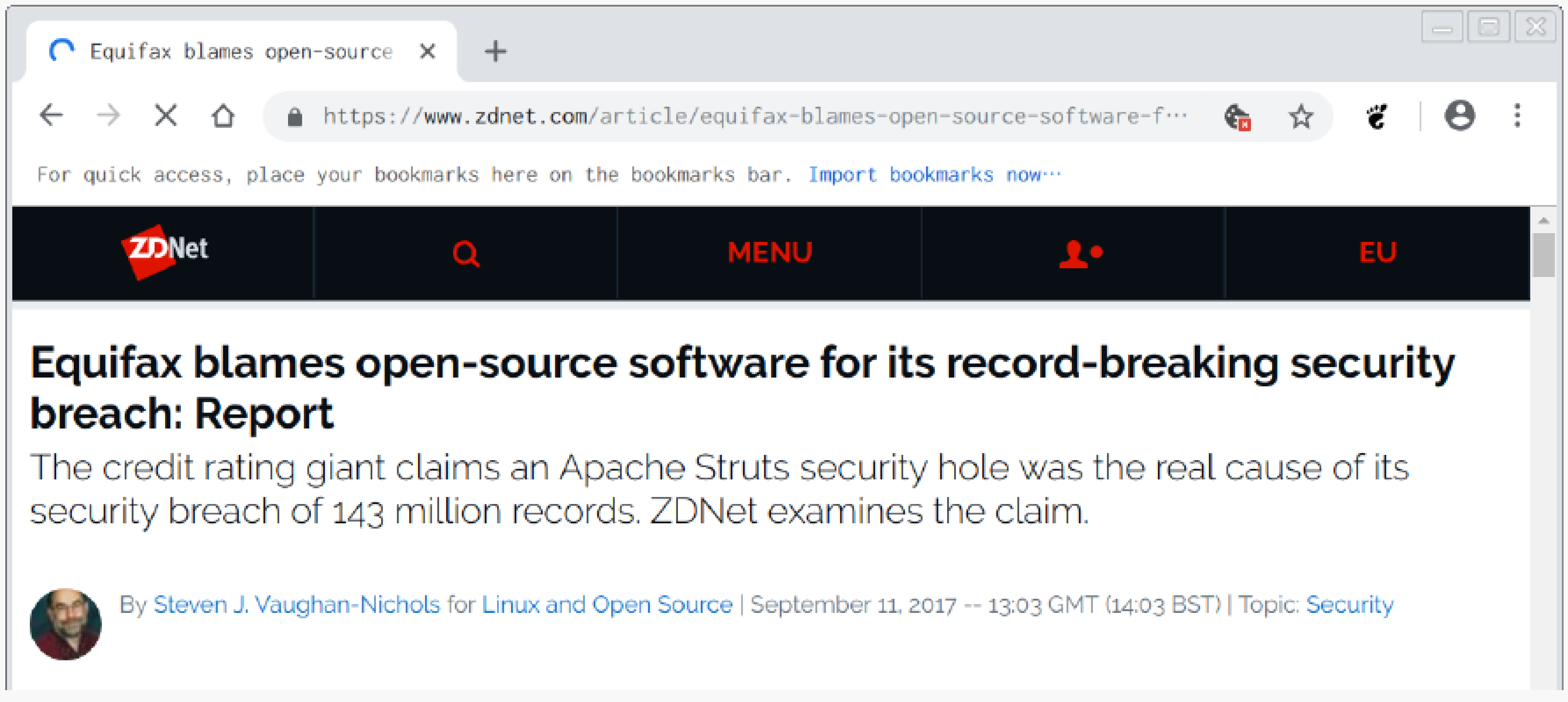
  ("HelloWorld" requires over 20 external libs)
- Only control over small fraction of source

Types of Third-party Software

	Proprietary Libraries Outsourcing Bespoke Software	Freeware	Free/Libre Open Source Software
Example	ILNumerics	Device Driver	Apache Tomcat
Upfront costs	High	Low	Low
Access for devs	Hard	Medium	Easy
Source Modification	Depends on contract	Impossible	Possible
Support contract	Easy	Hard	Medium

Vulnerabilities in Components

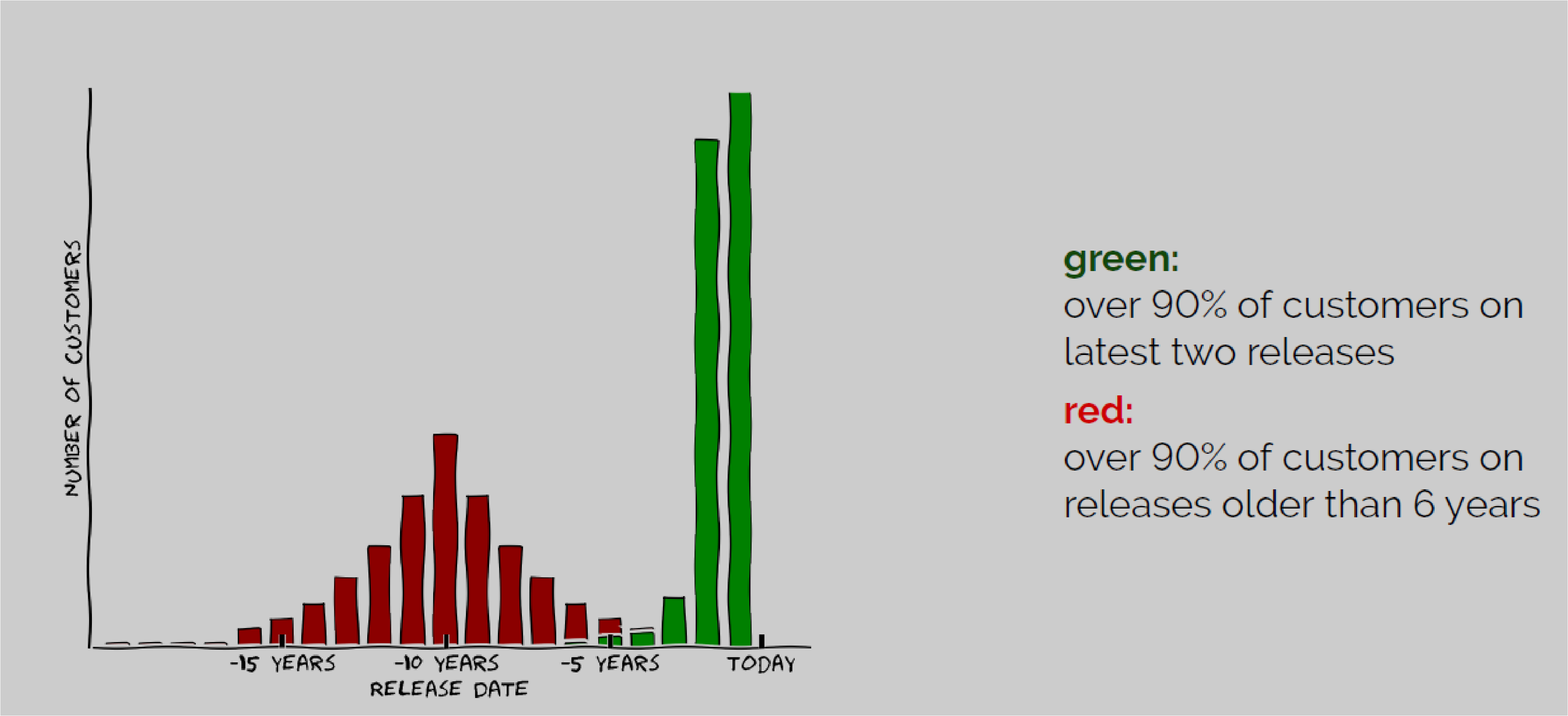
Equifax



Vulnerabilities in Components
Heartbleed

- Imagine
  - You are the Chief Product Security Officer for a software vendor
  - Your products consume many different external libraries
  - Different products consume different versions of the same library
- Now assume a severe vulnerability in an external library is published
  - How do you decide which products to fix first?
  - How do you decide how to fix (upgrade vs. down-port)?

What to Do?



- There seem to be an easy fix
    - ***Always use the latest version, i.e., update your dependencies as quickly as you can!***

Fast Upgrades Can Create Risks

The Register®
Biting the hand that feeds IT

DATA CENTRE | SOFTWARE | SECURITY | DEVOPS | BUSINESS | PERSONAL TECH | SCIENCE | EMERGENT TECH | BOOTNOTES | LECTURES

Software

How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript

Code pulled from NPM – which everyone was using

By Chris Williams, Editor in Chief 23 Mar 2016 at 01:24 167 SHARE ▼

Most read

Sure, Europe. Here's our Android suite without Search, Chrome apps. Now pay the Google tax

Leaked memo: No internet until you clean your bathroom, Ecuador told Julian Assange

Fed up with cloud giants ripping off its database, MongoDB forks new 'open-source license'

Thought Patch Tuesday was a load? You gotta

How Can We Minimize the Risk?
Design Your Application Securely

- Make the part of your application that needs to process critical data as small as possible

How Can We Minimize the Risk?

Design Your Application Securely

- Make the part of your application that needs to process critical data as small as possible

  (minimize the amount of code that you need to trust)

  - If a third-party library never touches confidential data, a vulnerability in that library is most likely not critical to you!

How Can We Minimize the Risk?
Select Your Dependencies Wisely

- Prefer projects
  - With an active development community
  - That use build systems, programming techniques that you are familiar with
  - That fit your support/release strategy
  - That follow best practices in secure development
    - Using security testing tools
    - Publishing regularly fixes and communicate openly about problems
    - Having coding guidelines (and following them)
  - Smaller components might have a smaller attack surface

How Can We Minimize the Risk?

Document and Monitor Your Dependencies

- Maintain a software inventory of all used component versions and where they are used
  - There are tools that can help (but they are not perfect), e.g.,
    - Your build system (e.g., paket, maven, npm)
    - OWASP dependency checker
  - They can also help to check license violations
  - Do not forget recursive (and hidden) dependencies
- Check daily for new published vulnerabilities
  - CVEs (NVD) cover only a small fraction, many projects do not publish CVEs (e.g., only list vulnerabilities on their own website, etc.)
  - Again, there are tools to help you, e.g.,
    - OWASP dependency checker
    - retire.js

How Can We Minimize the Risk?
Maintain Your Dependencies (and Applications)

- Upgrade components with security fixes and ship updates to customers

- Plan for efforts for down-porting patches

- Assign people responsible for maintaining components either

    - Locally in the development team, or
    - Create a global maintenance team
    - Alternatively, there are also companies offering commercial support for (nearly) any component

How Can We Minimize the Risk?
Harden Your Development Environment

- Check that you download the right component and, e.g.,
  - Not one with a similar name
  - Or some forked GitHub repository
- Ensure that downloads are using secure connections (https) and that signatures of signed packages are checked
- Use an own "artifactory" (package server) storing
  - The currently used version(s) of a component and
  - All previously used versions
- Only allow restricted network access from/to the build system/container

Secure Consumption of Third-party Libraries
Research Areas

- Analyze statically vulnerability reports and external software repository
  - Which versions (commit ranges) are vulnerable?
  - Which API calls are vulnerable?
  - How much did the API change between consumed version and the next fixed version?
- Derive fix recommendations
- Analyze consuming software (statically and/or dynamically)
  - Is the vulnerable API actually invoked?
  - Does the consuming software implement protection mechanisms?
  - Could the consuming software implement protection mechanisms?
- Generalize to global cost models
  - Maintenance of third-party libraries
  - Allow project managers to plan average development efforts

Reading List

- Brian Chess and Jacob West. Secure Programming with Static Analysis. Addison-Wesley Professional, First edition, 2007.
- Michael Felderer, Matthias Büchler, Martin Johns, Achim D. Brucker, Ruth Breu, and Alexander Pretschner. Security Testing: A Survey. Advances in Computers, 101:1–51, March 2016.
- Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated Whitebox Fuzz Testing. In Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008. The Internet Society, 2008.
- Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. Commun. ACM, 33(12):32–44, December 1990.
- Ruediger Bachmann and Achim D. Brucker. Developing Secure Software: A Holistic Approach to Security Testing. Datenschutz und Datensicherheit (DuD), 38(4):257–261, April 2014.

Thanks for your attention!

- Any questions or remarks?