Usable Security for Developers: A Nightmare

Achim D. Brucker | @adbrucker
About Me

- Security Expert/Architect at SAP SE
  - Member of the central security team, SAP SE (Germany)
    - Security Testing Strategist
  - Work areas at SAP included:
    - Defining the risk-based Security Testing Strategy
    - Evaluation of security testing tools (e.g., SAST, DAST)
    - Roll-out of security testing tools
    - Secure Software Development Life Cycle integration
    - ...
- Since December 2015:
  - Associate Professor, The University of Sheffield, UK
  - Head of the Software Assurance & Security Research Team
  - Available as consultancy & (research) collaborations

https://www.brucker.ch/
Outline

1. Security experts and developers
2. Secure programming can’t be that difficult …
3. The most common “fixes”
4. What we should do
70 years of software development

Since the late 1940ies, we

computer systems.
70 years of software development

Since the late 1940ies, we program, computer systems.
Since the late 1940ies, we program, debug, and computer systems.
Since the late 1940ies, we program, debug, and patch computer systems.

70 years of software development
70 years of software development

Since the late 1940ies, we
- program,
- debug, and
- patch
computer systems.

- we do not use punch cards anymore ...
We build software since 70 years
We build software since 70 years and still make the same old (security) mistakes
Usable Security for Developers: A Nightmare

Achim D. Brucker | @adbrucker
The common “silver bullet”: The SDLC

**Central security experts** (SDLC owner)
- Organizes security trainings
- Defines product standard “Security”
- Defines security testing strategy
- Validates products
- …

**Development teams**
- Select technologies
- Select development model
- Design and execute security testing plan
- …
Works nicely
Works nicely
in theory – let’s move to reality
Usable Security for Developers: A Nightmare

Achim D. Brucker | @adbrucker

Developer | Security Expert
Introducing the SDLC: View of the security experts

- The whiteboard is from the Microsoft’s security team
- I confess, I am guilty too: We also had a board with “embarrassing developers quotes”
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Kcm?</td>
<td>Will break everything</td>
<td>No testing</td>
<td>Someone else will fix it</td>
<td>NOT A SECURITY FEATURE</td>
<td>Try except? Will catch</td>
</tr>
<tr>
<td>D</td>
<td>IPsec?</td>
<td>Why wasn't</td>
<td>MANANA</td>
<td>App not working</td>
<td>THE END</td>
<td>Friends will stop it</td>
</tr>
<tr>
<td>E</td>
<td>SSL?</td>
<td>People will turn it off</td>
<td>What's the right?</td>
<td>NO NO NO</td>
<td>Fuzzing</td>
<td>Can't someone else fix it?</td>
</tr>
<tr>
<td>F</td>
<td>Behind the firewall</td>
<td>No custom</td>
<td>The real</td>
<td>Just a DOs</td>
<td>According to MSDN</td>
<td>WE DO NOT \ \ SUPPORT IT</td>
</tr>
<tr>
<td>G</td>
<td>That was someone else's</td>
<td>Someone would have to click on it</td>
<td>A bug has been found</td>
<td>The binary is not</td>
<td>It's not lovable</td>
<td>Can't overload because it's an unsigned</td>
</tr>
<tr>
<td>H</td>
<td>NLA?</td>
<td>Please make this bug hits less scary</td>
<td>The internet will fix it</td>
<td>The bug has been reported, don't bother</td>
<td>but...</td>
<td>Smart cards!!</td>
</tr>
<tr>
<td>I</td>
<td>The ISP is trusted</td>
<td>Only 15 sec</td>
<td>Perfect hit</td>
<td>Only local subnet</td>
<td>At we look up the same, server before connecting insecurely</td>
<td>It's ok, we use SECURITY JUMBO JUMBO</td>
</tr>
<tr>
<td>J</td>
<td>Office wire</td>
<td>it's old code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: The handwriting and symbols are unclear and may require additional context for accurate transcription.
SQL Injection:
I would never enter this!
SQL Injection:
I would never enter this!

Encryption:
We XOR-encrypted it
SQL Injection:
I would never enter this!

Encryption:
We XOR-encrypted it

Injection:
But that would be illegal!
SQL Injection:
I would never enter this!

Encryption:
We XOR-encrypted it

Injection:
But that would be illegal!

XSS (as a feature):
We can’t fix this, customers rely on it (sad but true)
Introducing the SDLC: View of the developers

- Experience security as “The Department of No”
- Confronted with a strange & complex language (there are over 1024 CWEs – and counting)
Example of unfriendly APIs: Buffer overflow

Let’s travel back in time

- Unix V7 (1979)
- Reading strings
- Gets returns a string of arbitrary length

Is there a secure use of gets?

---

> man gets
GETS(3S) GETS(3S)

NAME

gets, fgets - get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets(s)
char *s;
```

DESCRIPTION

Gets reads a string into s from the standard input stream stdin. The string is terminated by a newline character, which is replaced in s by a null character. Gets returns its argument.
Example of unfriendly APIs: Buffer overflow

Wait, let’s check the man page on a modern Unix/Linux:

<table>
<thead>
<tr>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>gets - get a string from standard input (DEPRECATED)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BUGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.</td>
</tr>
</tbody>
</table>
Example of unfriendly APIs: Buffer overflow

- OK, that’s sounds easy:

```c
void f() {
    char buf[20];
    gets(buf);
}
```
Example of unfriendly APIs: Buffer overflow

- OK, that’s sounds easy: Use `fgets(buf, n, stdin)` instead of `gets(buf)`:

```c
void f() {
    char buf[20];
    fgets(buf,20,stdin) // NOT: gets(buf);
}
```
Example of unfriendly APIs: Buffer overflow

- OK, that’s sounds easy: Use `fgets(buf, n, stdin)` instead of `gets(buf)`:

  ```c
  void f() {
      char buf[20];
      fgets(buf,20,stdin) // NOT: gets(buf);
  }
  ```

- Is this now secure?
Example of unfriendly APIs: Buffer overflow

- OK, that's sounds easy: Use `fgets(buf, n, stdin)` instead of `gets(buf)`: 
  
  ```c
  void f() {
      char buf[20];
      fgets(buf, 20, stdin) // NOT: gets(buf);
  }
  ```

- Is this now secure? No, `fgets` does **not** always null-terminate
Example of unfriendly APIs: Buffer overflow

- OK, that’s sounds easy: Use `fgets(buf, n, stdin)` instead of `gets(buf)`:

  ```c
  void f() {
    char buf[20];
    fgets(buf,20,stdin) // NOT: gets(buf);
  }
  ```

- Is this now secure? No, `fgets` does not always null-terminate
  we need to manually null terminate the buffer (and reserve space for the null character)

  ```c
  void f() {
    char buf[21];
    fgets(buf,20,stdin);
    buf[20]=‘\0’;
  }
  ```

- C-Programming has a lot in comming with (insurance) contracts: allways read the small print
Example of unfriendly APIs: Error handling

“Most OpenSSL functions will return an integer to indicate success or failure. Typically a function will \textbf{return} 1 \textbf{on success} or 0 \textbf{on error}. All return codes should be checked and handled as appropriate. Note that not all of the libcrypto functions return 0 for error and 1 for success. \textbf{There are exceptions which can trip up the unwary.} For example \textbf{if you want to check a signature with some functions you get} 1 \textbf{if the signature is correct}, 0 \textbf{if it is not correct} and -1 \textbf{if something bad happened} like a memory allocation failure.”

- Recall the common C convention:
  - 0 indicates success
  - any non-zero value indicates failure
Example of unfriendly APIs: Error handling

Which one is correct:

1. Consider

```c
if (some_verify_function())
/* signature successful */
```

2. Consider

```c
if (1 != some_verify_function())
/* signature successful */
```
Example of unfriendly APIs: Error handling

Which one is correct:

1. Consider

```c
if (some_verify_function())
/* signature successful */
```

2. Consider

```c
if (1 != some_verify_function())
/* signature successful */
```

The last one is correct
Example of unfriendly APIs: The Java 8 Crypto API

Just a nightmare:

- Many configurations to choose from
  - algorithm
  - mode of operation
  - padding scheme
  - right keys and sizes
  - …

- Most ciphers are outdated/broken. Only two can still be recommended
  - AES (symmetric)
  - RSA (asymmetric)

- Many providers use insecure defaults (e.g., ECB mode)

Using the Java crypto API, is already hard for somebody who understands crypto …
Example of unfriendly APIs: XSS (Java)

- Most Web Frameworks for Java do not provide input/output encoding as default
- Developers need to include third party encoding libraries (e.g., OWASP Java Encoder: https://github.com/OWASP/owasp-java-encoder)
- and add calls to the encoder manually:

```java
PrintWriter out = ....;
out.println("<textarea>"+Encode.forHtml(userData)+"</textarea>");
```

- You need to insert the **right** (there are many) encoder **each time**.
Common mitigations

- Provide training
  - Do we really expect that our developers understand all these details?
- Write (coding) guidelines
  - Guidelines without tool support are (mostly) worthless
- Use generic application security testing tools
  - without configuration, these tools are prone to both high false-positive rates and high false-negative rates
  - many tools are developed for security experts (and not for developers)
  - penetration tests

In their generality, these actions are often not very effective!
Security experts and developers need to work together to achieve the common goal: secure software!

(Disclaimer: security experts might need to learn how to code)!

Think positive: security enables developers to produce high-quality and secure software!

- Start early in the development:
  - Select frameworks and/or programming languages that are secure by design
  - Develop custom APIs-Wrappers that are easy to use and require only little security knowledge
  - To consider
    - Configure your DAST/IAST/SAST tool to support your custom APIs
    - In the fix recommendations of your DAST/IAST/SAST tool, point developers to the recommended frameworks
    - If you develop APIs, make your examples secure by default
If you do not support your developers, they will seek for help elsewhere!
Let’s close with a good example: Modern Rails

- Modern versions of Rails are pretty secure by default
- Input/output encoding is enabled by default and, in exceptional cases, needs to be disabled explicitly:
  ```html
  <%= account.balance.html_safe %>
  ```
  (one can argue, if html_safe is a good name denoting un-sanitized (trusted) channels)
- Suddenly, a simple grep becomes a powerful static analysis tool
Call for action

Let’s build framework and APIs are easy to use securely!
Call for action

Let’s build framework and APIs are easy to use securely!
Call for action

Let’s build framework and APIs are easy to use securely!
Call for action

Let’s build framework and APIs are easy to use securely!
Call for action

Let’s build framework and APIs are easy to use securely!
Thank you for your attention!
Any questions or remarks?
Document Classification and License Information

© 2018 LogicalHacking.com, Achim D. Brucker | @adbrucker.

This presentation is classified as Public (CC BY-NC-ND 4.0):
Except where otherwise noted, this presentation is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International Public License (CC BY-NC-ND 4.0).