OWASP
Proactive Controls 2018

Boston Code Camp
November 17, 2018
Robert Hurlbut
@RobertHurlbut

Bank of America

BANK OF AMERICA | MERRILL LYNCH | U.S. TRUST | BANK OF AMERICA MERRILL LYNCH

Who am I?

Robert Hurlbut
SVP, Threat Modeling Architect / Lead
Cyber Security Technology
Bank of America

Boston Code Camp 30 - Thanks to our Sponsors!

OWASP Proactive Controls v3 (2018)

About OWASP Top 10 Proactive Controls

- The controls are intended to provide initial awareness around building secure software.
- The document provides a good foundation of topics to help drive introductory software security developer training.
- These controls should be used consistently and thoroughly throughout all applications.

Project Leaders & Contributors

Project Leaders

Jim Manico
Jim Bird
Katy Anton

Contributors

Chris Romeo
Dan Anderson
David Cybuck
Dave Ferguson
Josh Grossman
Osama Elnaggar
Rick Mitchell

OWASP Top Ten Proactive Controls v3 (2018)

C1 Define Security Requirements	C2 Leverage Security Frameworks and Libraries	C3 Secure Database Access	C4 Encode and Escape Data
C5 Validate All Inputs	C6 Implement Digital Identity	C7 Enforce Access Control	C8 Protect Data Everywhere
	C9 Implement Security Logging and Monitoring	C10 Handle All Errors and Exceptions	

OWASP Proactive Controls 2018

Find the OWASP Proactive Controls 2018 document at the OWASP Proactive Controls Project page:

https://www.owasp.org/index.php/OWASP_Proactive_Controls

C1: Define Security Requirements

- Software Development Lifecycle (SDLC)
- OWASP Application Security Verification Standard (ASVS) v.3.0.1 (v.4 in works now)
  - Catalog of available security requirements and verification criteria.
  - Level 1 – Baseline (82 controls)
  - Level 2 – Standard (139 controls)
  - Level 3 – Comprehensive (154 controls)
  - All helpful for writing tests (unit tests and penetration tests)
  - User stories and misuse cases
- Also – OWASP Mobile ASVS

C1: Define Security Requirements, continued

Requirements

#	Description	1	2	3	Since
1.1	Verify that all application components are identified and are known to be needed.	✓	✓	✓	1.0
1.2	Verify that all components, such as libraries, modules, and external systems, that are not part of the application but that the application relies on to operate are identified.		✓	✓	1.0
1.3	Verify that a high-level architecture for the application has been defined.		✓	✓	1.0
1.4	Verify that all application components are defined in terms of the business functions and/or security functions they provide.			✓	1.0
1.5	Verify that all components that are not part of the application but that the application relies on to operate are defined in terms of the functions, and/or security			✓	1.0

C1: Define Security Requirements, continued

ASVS Resources

OWASP Application Security Verification Standard Project

- https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project

OWASP Mobile Application Security Verification Standard Project

- https://www.owasp.org/images/6/61/MASVS_v0.9.4.pdf

C2: Leverage Security Frameworks and Libraries

- Don't reinvent the wheel
  - Lots of coding libraries and software frameworks
  - Use native security features of frameworks
  - Stay up to date!
- Keep inventory catalog of third party libraries
- Use tools like OWASP Dependency Check, Retire.JS to identify project dependencies - check for known, publicly disclosed vulnerabilities for all third party code (ideally, automate checks every build)
  - OWASP Dependency Check
    - https://www.owasp.org/index.php/OWASP_Dependency_Check
  - Retire.JS
    - https://retirejs.github.io/retire.js/

C3: Secure Database Access

The Perfect Password

X' or '1'='1' --

- ✓ Upper
- ✓ Lower
- ✓ Number
- ✓ Special
- ✓ Over 16 characters

C3: Secure Database Access, continued

The Perfect Email Address

`john.doe'or'1'!='@acme.com`

- ✓ RFC Compliant
- ✓ Should validate as legit email
- ✓ It's active now if you want to try
- ✓ Unsafe for SQL

C3: Secure Database Access, continued

SQL Injection attack from valid data:

```
select id,ssn,cc,mmn from customers where
email='$email'
```

```
$email = john.doe'or'1'!='@acme.com
```

```
select id,ssn,cc,mmn from customers where
email='john.doe'or'1'!='@acme.com'
```

C3: Secure Database Access, continued

- Secure queries
  - Parametrized queries
  - Certain parts of a query (i.e. table names) are not able to be parametrized – varies by vendor
- Secure configuration
  - Not all DB installations are “secure by default”
- Secure authentication
  - Watch what authentication is used to connect to the DB (i.e. not SA/no password)
- Secure communication
  - Secure (i.e. encrypted) channels

C3: Secure Database Access, continued

Caution

- One SQL Injection can lead to complete data loss. Be rigorous in keeping SQL Injection out of your code. There are several other forms of injection to consider as well.

Verify

- Code review and static analysis do an excellent job of discovering SQL Injection in your code

Guidance

- *http://bobby-tables.com/*
- *https://www.owasp.org/index.php/Query_Parameterization_Cheat_Sheet*
- *https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet*

C4: Encode and Escape Data

- Encoding and escaping - defensive techniques meant to stop injection attacks
- Encoding (commonly called "Output Encoding") - translating special characters into some different but equivalent form no longer dangerous in the target interpreter
  - "<" character into the < string when writing to an HTML page
- Escaping involves adding a special character before the character/string to avoid being misinterpreted
  - Adding "\" character before a """ (double quote) makes sure interpreted as text and not as closing a string

C4: Encode and Escape Data, continued

Review: XSS Defense Summary

Data Type	Context	Defense
String	**HTML Body/Attribute**	**HTML Entity Encode**
String	**JavaScript Variable**	**JavaScript Hex Encoding**
String	**GET Parameter**	**URL Encoding**
String	**Untrusted URL**	**URL Validation, avoid JavaScript: URLs, Attribute Encoding, Safe URL Verification**
String	**CSS**	**CSS Hex Encoding**
HTML	Anywhere	HTML Sanitization (Server and Client Side)
Any	DOM	Safe use of JS API's
Untrusted JavaScript	Any	Sandboxing and Deliver from Different Domain
JSON	Client Parse Time	JSON.parse() or json2.js
JSON	Embedded	JSON Serialization

C4: Encode and Escape Data, continued

Caution

- XSS defense as a total body of knowledge is extremely complicated. Continually be mindful of good XSS defense engineering

Verify

- SAST and DAST security tools are both good at XSS discovery.

Guidance

- *https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet*
- *https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet*
- *https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet*

C4: Encode and Escape Data, continued

Other kinds of injection:

- Command Injection
  - https://www.owasp.org/index.php/Command_Injection
- LDAP Injection
  - https://www.owasp.org/index.php/LDAP_Injection_Prevention_Cheat_Sheet
- Injection Protection in Java
  - https://www.owasp.org/index.php/Injection_Prevention_Cheat_Sheet_in_Java

C5: Validate All Inputs

Applications should check data is both ***syntactically*** and ***semantically*** valid before using it in any way.

Syntax validity -> data is in expected form.

- For example, an application may allow a user to select a four-digit "account ID" - the application checks data entered by user is exactly four digits in length and consists only of numbers

Semantic validity -> the data is within an acceptable range for the given application functionality and context

- For example, in a date range, a start date must be before the end date

C5: Validate All Inputs, continued

Tools for various environments:

OWASP HTML Sanitizer Project (Java)
- *https://code.google.com/p/owasp-java-html-sanitizer/wiki/AttackReviewGroundRules*

Java
- http://hibernate.org/validator/
- http://beanvalidation.org/

PHP's filter functions
- https://secure.php.net/manual/en/filter.examples.validation.php

Ruby on Rails
- http://edgeapi.rubyonrails.org/classes/ActionView/Helpers/SanitizeHelper.html

JavaScript
- https://github.com/cure53/DOMPurify

Python
- https://pypi.python.org/pypi/bleach

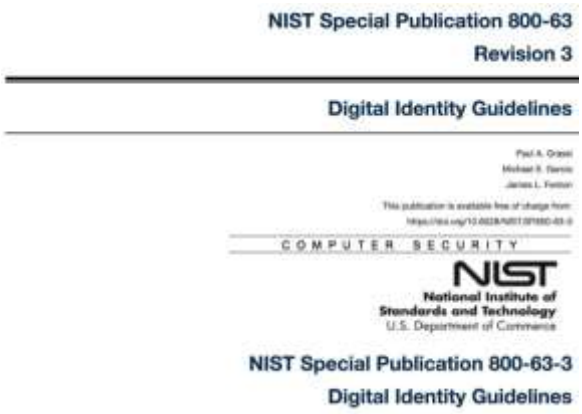
.NET
- https://github.com/mganss/HtmlSanitizer

Ruby on Rails
- https://rubygems.org/gems/loofah

C6: Implement Digital Identity

Applying authentication – verification an entity is *who it claims to be*

C6: Implement Digital Identity, continued



C6: Implement Digital Identity, continued

Password storage best practices

1. Do not limit the characters or length of user password
2. Use a modern password policy scheme
3. Hash the password using SHA2-512 or another strong hash
4. Combine a credential-specific random and unique salt to the hash
5. Use BCRYPT, SCRYPT, PBKDF2 or Argon2 on the combined salt and hash
6. Store passwords as an HMAC + good key management as an alternative

C6: Implement Digital Identity, continued

Authentication Cheat Sheet

- https://www.owasp.org/index.php/Authentication_Cheat_Sheet

Password Storage Cheat Sheet

- https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet

Forgot Password Cheat Sheet

- https://www.owasp.org/index.php/Forgot_Password_Cheat_Sheet

Session Management Cheat Sheet

- https://www.owasp.org/index.php/Session_Management_Cheat_Sheet

ASVS AuthN and Session Requirements

NIST 800-63-3 Digital Authentication Guidelines

- https://pages.nist.gov/800-63-3/sp800-63-3.html

C7: Enforce Access Control

Apply authorization – verification what an entity is *entitled to do*

C7: Enforce Access Control, continued

Access Control Anti-Patterns

- Hard-coded role checks in application code
- Lack of centralized access control logic
- Untrusted data driving access control decisions
- Access control that is “open by default”
- Lack of addressing horizontal access control in a standardized way (if at all)
- Access control logic that needs to be manually added to every endpoint in code
- Access Control that is “sticky” per session
- Access Control that requires per-user policy

C7: Enforce Access Control, continued

Access Control Design

- Consider **attribute based access control** design (ABAC) over role based access control (RBAC)
- Build **proper data contextual access control methodologies** - build database that understands which user may access which individual object
- Build access control design not just for one feature but for your whole application
- Consider adding a simple ownership relationship to data items so only data owners can view that data

C7: Enforce Access Control, continued

Caution

- Good access control is **hard to add to an application late in the lifecycle**. Work hard to get this right up front early on – i.e. *threat modeling*.

Verify

- Turnkey security tools cannot verify access control since tools are not aware of your applications policy. Be prepared to do security unit testing and manual review for access control verification.

Guidance

- *https://www.owasp.org/index.php/Access_Control_Cheat_Sheet*
- *http://nvlpubs.nist.gov/nistpubs/specialpublications/NIST.sp.800-162.pdf*

C8: Protect Data Everywhere

Encrypt Data in Transit

What benefits does HTTPS provide?

Confidentiality: Spy cannot view your data

Integrity: Spy cannot change your data

Authenticity: Server you are visiting is the right one

Performance: HTTPS is much more performant than HTTP on modern processors

HTTPS configuration best practices

- *https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet*
- *https://www.ssllabs.com/projects/best-practices/*

C8: Protect Data Everywhere, continued

Encrypt Data in Transit

HSTS (Strict Transport Security)

- http://www.youtube.com/watch?v=zEV3HOuM_Vw

Forward Secrecy

- https://whispersystems.org/blog/asynchronous-security/

Certificate Creation Transparency

- http://certificate-transparency.org

Certificate Pinning

- https://www.owasp.org/index.php/Pinning_Cheat_Sheet

Browser Certificate Pruning

C8: Protect Data Everywhere, continued

Encrypt Data at Rest

Be sure to use good, well-tested cryptographic methods (i.e. don't roll your own!)

Some libraries:

- Google Tink
  - *https://github.com/google/tink*
- Libsodium
  - *https://www.gitbook.com/book/jedisct1/libsodium/details*

Use a form of secrets management to protect application secrets and keys

- *https://www.vaultproject.io/*

C8: Protect Data Everywhere, continued

Caution

- Protecting sensitive data at rest and in transit is painfully tough to build and maintain, especially for intranet infrastructure. Commit to long term plans to continually improve in this area. Consider enterprise class solutions here.

Verify

- Bring in heavy-weight resources to verify your cryptographic implementations, especially at rest.

Guidance

- https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet
- https://www.ssllabs.com/projects/documentation/
- https://www.owasp.org/index.php/Cryptographic_Storage_Cheat_Sheet

C9: Implement Security Logging and Monitoring

Proper Application Security Logging

- Use common/standard logging approach to facilitate correlation and analysis
  - Logging framework : SLF4J with Logback, Apache Log4j2, Log4Net
- Perform encoding on untrusted data : protection against Log injection attacks!
- Be careful about logging sensitive data
- Consider using a logging abstraction layer that allows you to log events with security metadata
- Work with incident response teams to ensure proper security logging

C9: Implement Security Logging and Monitoring, continued

Detection Points Examples

- Input validation failure server side when client side validation exists
- Input validation failure server side on non-user editable parameters such as hidden fields, checkboxes, radio buttons or select lists
- Forced browsing to common attack entry points
- Honeypot URL (e.g. a fake path listed in robots.txt like e.g. /admin/secretlogin.aspx)

C9: Implement Security Logging and Monitoring, continued

Secure Logging Design

- **Encode** and **validate** any dangerous characters before logging to prevent **log injection** or **log forging** attacks
- Do not log sensitive information. For example, do not log password, session ID, credit cards or social security numbers
- **Protect log integrity** – consider permission of log files and log changes audit
- Forward logs from distributed systems to a central, secure logging service for **centralized monitoring**

C9: Implement Security Logging and Monitoring, continued

Caution

- Be sure developers and security teams work together to ensure good security logging

Verify

- Verify proper security events are getting logged

Guidance

- https://www.owasp.org/index.php/Category:OWASP_Logging_Project
- https://www.owasp.org/index.php/OWASP_Security_Logging_Project
- https://www.owasp.org/index.php/Logging_Cheat_Sheet

C10: Handle All Errors and Exceptions

Best Practices

- Manage exceptions in a **centralized manner.**
- **Avoid duplicated try/catch** blocks in the code.
- Ensure that all **unexpected behaviors are correctly handled** inside the application.
- Ensure error messages displayed to users do not leak **critical data**, but are still verbose enough to explain the issue to the user.
- Ensure that exceptions are logged in a way that gives enough information for Q/A, forensics or incident response teams to **understand the problem**.
- Consider the RESTful mechanism of using standard HTTP response codes for errors **instead of creating your own error code system**.

Special thanks to work by the OWASP Proactive Controls v3 team – many slides in this presentation were first made available publicly in this deck:

https://www.owasp.org/images/1/13/OWASP_Top_Ten_Proactive_Controls_v3.pptx

Questions?

@RobertHurlbut

Thank you!

Slides: https://roberthurlbut.com/r/BCC30PC18