IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p- ISSN: 2278-8727Volume 14, Issue 6 (Sep. - Oct. 2013), PP 14-22
www.iosrjournals.org

Secure Development - Web Application Security.

Sayyad Arif Ulla
Assistant Professor of Computer Science, Government Degree College,Sindhanur-584128.

Abstract: *In Current scenario, many Web applications go through rapid development phases like adopting agile methodology of development with very short turnaround time, making it difficult to identify and eliminate vulnerabilities.*

This paper provides analysis of requirement of Secure Development and Web application security assessment mechanisms in order to identify poor coding practices that render Web applications vulnerable to attacks such as SQL injection and cross-site scripting.

This paper also list out the different categories of vulnerability with the small examples along with prevention guidance and a sample of research by different vendors about the State of Software Security Report and Web Application Vulnerability Statistics of 2012.

This paper reviews need of secure development, resource s available for creating secure Web applications. These resources ranges from the security features of the development, to automated tools evaluating an existing Web application, to Web sites dedicated to all facets of Web application security.

In Web application security, making one single mistake can lead to many unwanted flaws. By using the different resources available, the risk of the applications to be vulnerable can be reduced to an acceptable level. In addition, some risk can be avoided at the very beginning of the project life cycle when the requirements for the system are defined.

I. Introduction:

The exploitation of web application vulnerabilities and vulnerability prone applications are some of the reason of enterprise data loss, hacking and malfunctioning of the intended purpose, and even in the wake of numerous high profile and well publicized breaches, many organizations have failed to address the most common application flaws making them vulnerability prone.

As the web becomes increasingly complex, web applications become more sophisticated and dynamic. One of the most important ways that web applications have become complex is in how they use input data. Web pages are no longer static; they contain dynamic content from sources that may be trusted, untrusted etc. There are many places that this type of data can come from: user input, advertisements, or widgets and many more. These sources of data have led to a class of attacks know as content injection attacks. In these attacks, an attacker is able to place malicious content on a page and make it act as if it came from the developer. This can lead to cross-site scripting attacks, .0 malicious information.

Requirement to counter these types of attacks, developers should implement web application security policies.

Web applications remain most vulnerable, with hacking still on the increase, from organized criminal groups, amateurs and political activists. Complex technology, growing adoption of web technology have enhanced the opportunity for hackers to exploit vulnerabilities. The consequences of a compromised web application can go way beyond the web server: a number of high-profile attacks with prestigious companies caused millions USD in losses. All organizations are potential victims. To protect themselves they should form long-term partnerships with reputable security companies providing individual solutions that will optimize web application security.

Secure Development

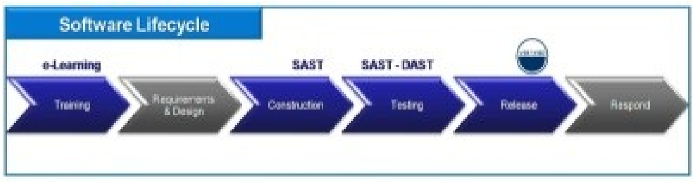
The Importance of Secure Development

With the vast amount of threats that constantly pressure companies and governments, it is important to ensure that the software applications these organizations utilize are completely secure. Secure development is a practice to ensure that the code and processes that go into developing applications are as secure as possible. Secure development entails the utilization of several processes, including the implementation of a Secure Development Lifecycle (SDLC) and secures coding itself.

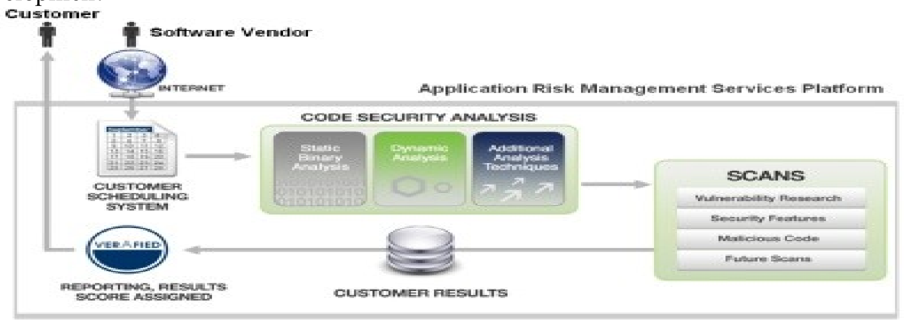
Secure Development Lifecycle

Integrating security practices into the software development lifecycle and verifying security of internally developed applications before they are deployed can help mitigate risk from internal and external

sources. Using scan tools to test the security of applications helps customers implement a secure development program in a simple and cost-effective way.
The Security Development Lifecycle (SDL) is a software development security assurance process consisting of security practices grouped by six phases: training, requirements & design, construction, testing, release, and response.



One of the important steps in Secure development is integrating testing tools and services of scan tools into the software development lifecycle. These tools allow developers to model an application, scan the code, check the quality and ensure that it meets regulations. Automated secure development testing tools help developers find and fix security issues.
Secure development can be incorporated into both a traditional software development lifecycle and the rapid pace agile development



Vulnerability Details

Brief description about Top 10 Vulnerabilities identified for 2013:
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 unauthorized data.
Example Scenarios
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 that enable a complete takeover of the database and possibly even the server hosting the database.

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. Be careful of APIs, such as stored procedures, that are parameterized, but can still introduce 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 provides many of these escaping routines.
3. Positive or "white list" input validation with appropriate canonicalization is also recommended, but is not a complete defense as many applications require special characters in their input

2. Broken Authentication and Session Management
Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, session tokens, or exploit other implementation flaws to assume other users' identities.
Example Scenarios

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

How Do I Prevent Broken Authentication and Session Management?
The primary recommendation for an organization is to make available to developers:

1. **A single set of strong authentication and session management controls.**
2. Strong efforts should also be made to avoid XSS flaws which can be used to steal session IDs

3. Cross-Site Scripting (XSS)
XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping. 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.
Example 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 their browser to:

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

This 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 also use XSS to defeat any automated CSRF defense the application might employ. See A8 for info on CSRF. Detection of most XSS flaws is fairly easy via testing or code analysis
How Do I Prevent Cross-Site Scripting (XSS)?
Preventing XSS requires keeping untrusted data separate from active browser content.

1. The preferred option is to properly escape all untrusted data based on the HTML context (body, attribute, JavaScript, CSS, or URL) that the data will be placed into. See the OWASP XSS Prevention Cheat Sheet for details on the required data escaping techniques.
2. Positive or "whitelist" input validation is also recommended as it helps protect against XSS, but is not a complete defense as many applications require special characters in their input. Such validation should, as much as possible, validate the length, characters, format, and business rules on that data before accepting the input.
3. For rich content, consider auto-sanitization libraries like OWASP's AntiSamy.

4. Insecure Direct Object References
A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.
Example Scenarios
The application uses unverified data in a SQL call that is accessing account information:

```
String query = "SELECT * FROM accts WHERE account = ?"; PreparedStatement pstmt =
connection.prepareStatement(query , … ); {{red|pstmt.setString( 1, request.getParameter("acct"));}} ResultSet
results = pstmt.executeQuery( );
```

The attacker simply modifies the 'acct' parameter in their browser to send whatever account number they want. If not verified, the attacker can access any user's account, instead of only the intended customer's account.

```
http://example.com/app/accountInfo?acct=<span style="color:red;">notmyacct</span>
```

How Do I Prevent Insecure Direct Object References?
Preventing insecure direct object references requires selecting an approach for protecting each user accessible object (e.g., object number, filename):

1. Use per user or session indirect object references. This prevents attackers from directly targeting unauthorized resources. For example, instead of using the resource's database key, a drop down list of six resources authorized for the current user could use the numbers 1 to 6 to indicate which value the user selected. The application has to map the per-user indirect reference back to the actual database key on the server. OWASP's ESAPI includes both sequential and random access reference maps that developers can use to eliminate direct object references.
2. Check access. Each use of a direct object reference from an untrusted source must include an access control check to ensure the user is authorized for the requested object

5. Security Misconfiguration

Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. All these settings should be defined, implemented, and maintained as many are not shipped with secure defaults. This includes keeping all software up to date.

Example Scenarios

<u>Scenario #1:</u> 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.

<u>Scenario #2:</u> Directory listing is not disabled on your server. Attacker discovers she can simply list directories to find any file. Attacker finds and downloads all your compiled Java classes, which she reverses to get all your custom code. She then finds a serious access control flaw in your application.

<u>Scenario #3:</u> App server configuration allows stack traces to be returned to users, potentially exposing underlying flaws. Attackers love the extra information error messages provide.

<u>Scenario #4:</u> App server comes with sample applications that are not removed from your production server. Said sample applications have well known security flaws attackers can use to compromise your server.

How Do I Prevent Security Misconfiguration?

The primary recommendations are to establish all of the following:

1. A repeatable hardening process that makes it fast and easy to deploy another environment that is properly locked down. Development, QA, and production environments should all be configured identically. This process should be automated to minimize the effort required to setup a new secure environment.
2. A process for keeping abreast of and deploying all new software updates and patches in a timely manner to each deployed environment. This needs to include all code libraries as well (see new A9).
3. A strong application architecture that provides good separation and security between components.
4. Consider running scans and doing audits periodically to help detect future misconfigurations or missing patches.

6. Sensitive Data Exposure

Many web applications do not properly protect sensitive data, such as credit cards, tax ids, and authentication credentials. Attackers may steal or modify such weakly protected data to conduct identity theft, credit card fraud, 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.

Example Scenarios

<u>Scenario #1:</u> An application encrypts credit card numbers in a database using automatic database encryption. However, this means it also decrypts this data automatically when retrieved, allowing an SQL injection flaw to retrieve credit card numbers in clear text. The system should have encrypted the credit card numbers using a public key, and only allowed back-end applications to decrypt them with the private key.

<u>Scenario #2:</u> A site simply doesn't use SSL for all authenticated pages. Attacker simply monitors network traffic (like an open wireless network), and steals the user's session cookie. Attacker then replays this cookie and hijacks the user's session, accessing all their private data.

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

How Do I Prevent Sensitive Data Exposure?

The full perils of unsafe cryptography, SSL usage and for all sensitive data, do all of the following, at a minimum:

1. Considering the threats you plan to protect this data from (e.g., insider attack, external user), make sure you encrypt all sensitive data at rest and in transit in a manner that defends against these threats.
2. Don't store sensitive data unnecessarily. Discard it as soon as possible. Data you don't have can't be stolen.
3. Ensure strong standard algorithms and strong keys are used, and proper key management is in place.

Ensure passwords are stored with an algorithm specifically designed for password protection, such as <u>bcrypt</u>, <u>PBKDF2</u>, or <u>scrypt</u>.

1. Disable autocomplete on forms collecting sensitive data and disable caching for pages displaying sensitive data.

7. Missing Function Level Access Control

Virtually all web applications verify function level access rights before making that functionality visible in the UI. However, applications need to perform the same access control checks on the server when each function is accessed. If requests are not verified, attackers will be able to forge requests in order to access unauthorized functionality.

Example Scenarios

Scenario #1: The attacker simply force browses to target URLs. The following URLs require authentication. Admin rights are also required for access to the "admin_getappInfo" page.

```
http://example.com/app/getappInfo http://example.com/app/admin_getappInfo
```

If an unauthenticated user can access either page, that's a flaw. If an authenticated, non-admin, user is allowed to access the "admin_getappInfo" page, this is also a flaw, and may lead the attacker to more improperly protected admin pages.

Scenario #2: A page provides an 'action 'parameter to specify the function being invoked, and different actions require different roles. If these roles aren't enforced, that's a flaw.

How Do I Prevent Missing Function Level Access Control?

Your application should have a consistent and easily analyzable authorization module that is invoked from all your business functions. Frequently, such protection is provided by one or more components external to the application code.

1. Think about the process for managing entitlements and ensure you can update and audit easily. Don't hard code.
2. The enforcement mechanism(s) should deny all access by default, requiring explicit grants to specific roles for access to every function.
3. If the function is involved in a workflow, check to make sure the conditions are in the proper state to allow access.

NOTE: Most web applications don't display links and buttons to unauthorized functions, but this "presentation layer access control" doesn't actually provide protection. You must also implement checks in the controller or business logic.

8. Cross-Site Request Forgery (CSRF)

A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

Example Scenarios

The application allows a user to submit a state changing request that does not include anything secret. For example:

```
http://example.com/app/transferFunds?amount=1500&destinationAccount=4673243243
<img src="<span style="color: red;">http://example.com/app/transferFunds?amount=1500&destinationAccount=attackersAcct#</span>" width="0" height="0" />
```

So, the attacker constructs a request that will transfer money from the victim's account to their account, and then embeds this attack in an image request or iframe stored on various sites under the attacker's control like so: If the victim visits any of the attacker's sites while already authenticated to example.com, these forged requests will automatically include the user's session info, authorizing the attacker's request.

How Do I Prevent Cross-Site Request Forgery (CSRF)?

Preventing CSRF usually requires the inclusion of an unpredictable token in each HTTP request. Such tokens should, at a minimum, be unique per user session.

1. The preferred option is to include the unique token in a hidden field. This causes the value to be sent in the body of the HTTP request, avoiding its inclusion in the URL, which is subject to exposure.
2. The unique token can also be included in the URL itself, or a URL parameter. However, such placement runs the risk that the URL will be exposed to an attacker, thus compromising the secret token.
3. Requiring the user to reauthenticate, or prove they are a user (e.g., via a CAPTCHA) can also protect against CSRF.

9. Using components with Known Vulnerabilities
Vulnerable components, such as libraries, frameworks, and other software modules almost always run with full privilege. So, if exploited, they can cause serious data loss or server takeover. Applications using these vulnerable components may undermine their defenses and enable a range of possible attacks and impacts.
Example Scenarios
Component vulnerabilities can cause almost any type of risk imaginable, from the trivial to sophisticated malware designed to target a specific organization. Components almost always run with the full privilege of the application, so flaws in any component can be serious, The following two vulnerable components were downloaded 22m times in 2011.

- Apache CXF Authentication Bypass – By failing to provide an identity token, attackers could invoke any web service with full permission.
- Spring Remote Code Execution– Abuse of the Expression Language implementation in Spring allowed attackers to execute arbitrary code, effectively taking over the server.

Every application using either of these vulnerable libraries is vulnerable to attack as both of these components are directly accessible by application users. Other vulnerable libraries, used deeper in an application, may be harder to exploit.
How Do I Prevent Using Components with Known Vulnerabilities?
One option is not to use components that you didn't write. But realistically, the best way to deal with this risk is to ensure that you keep your components up-to-date. Many open source projects (and other component sources) do not create vulnerability patches for old versions. Instead, most simply fix the problem in the next version. Software projects should have a process in place to:

1. Identify the components and their versions you are using, including all dependencies. (e.g., the versions plugin).
2. Monitor the security of these components in public databases, project mailing lists, and security mailing lists, and keep them up-to-date.
3. Establish security policies governing component use, such as requiring certain software development practices, passing security tests, and acceptable licenses.

10. Un validated Redirects and Forwards
Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.
Example Scenarios
Scenario #1: The application has a page called "redirect.jsp" which takes a single parameter named "url". The attacker crafts a malicious URL that redirects users to a malicious site that performs phishing and installs malware.

```
http://www.example.com/redirect.jsp?url=evil.com
```

Scenario #2: The application uses forwards to route requests between different parts of the site. To facilitate this, some pages use a parameter to indicate where the user should be sent if a transaction is successful. In this case, the attacker crafts a URL that will pass the application's access control check and then forwards the attacker to an administrative function that she would not normally be able to access.

```
http://www.example.com/boring.jsp?fwd=admin.jsp
```

How Do I Prevent Unvalidated Redirects and Forwards?
Safe use of redirects and forwards can be done in a number of ways:

1. Simply avoid using redirects and forwards.
2. If used, don't involve user parameters in calculating the destination. This can usually be done.
3. If destination parameters can't be avoided, ensure that the supplied value is valid, and authorized for the user.

   It is recommended that any such destination parameters be a mapping value, rather than the actual URL or portion of the URL, and that server side code translate this mapping to the target URL.

   Applications can use ESAPI to override the sendRedirect() method to make sure all redirect destinations are safe.

Avoiding such flaws is extremely important as they are a favorite target of phishers trying to gain the user's trust.

Analysis and Reports:
Veracode State of Software Security Report:
- Veracode, Inc., the leader in cloud-based application security testing, today released the second feature supplement of its annual **State of Software Security Report (SoSS),** for the first time analyzing third party software security testing metrics. The data indicates that despite increasing security risks from third party and externally developed software, few enterprises currently have formal testing programs in place. However, there are signs that more organizations are beginning to recognize and address the security risks associated with externally developed applications.

"The widespread adoption of third-party apps and use of external developers in enterprises brings increased risk," said Chris Eng, vice president of research, Veracode. "In fact, a typical enterprise has an average of 600 mission-critical applications, about 65% of which are developed externally, leaving companies increasingly vulnerable to the security risks found in these apps. We are beginning to see signs that enterprises are recognizing and addressing these risks. However, organizations still assume too much risk when trusting their third-party software suppliers to develop applications that meet industry and organizational standards. There is still much more work to be done to adequately secure the software supply chain."
The supplement found that some of the most dangerous security flaws in existence, such as SQL injection and Cross Site Scripting, are among the most prevalent vulnerabilities in third-party vendor applications. The report also showed that while a programmatic approach to software security testing can greatly help enterprises and their vendors mitigate these flaws, few organizations have formal programs in place to manage and secure the software supply chain.

Key findings:
Currently few enterprises have vendor application security testing programs in place, but the volume of assessments within organizations is growing
1.Less than one in five enterprises have requested a code-level security test from at least one vendor

2. However, the volume of vendor supplied software or application assessments continues to grow with a 49% increase from the first quarter of 2011 to the second quarter of 2012

3. **There is a Gap Between Enterprise Standard and Industry Standard Compliance**

4 . 38% of vendor supplied applications complied with enterprise-defined policies vs. 10% with the OWASP Top Ten and 30% with CWE/SANS Top 25 industry-defined standards.

5. **Some of the most dangerous vulnerabilities in vendor applications are also the most prevalent** .

6. Four of the top five flaw categories for web applications are also among the OWASP Top 10 most dangerous flaws and five of the top six flaw categories for non-web applications appear on the CWE/SANS Top 25 list of most dangerous flaws.

7. SQL injection and cross-site scripting affect 40 percent and 71 percent of vendor-supplied web application versions, respectively.

8. Only 10 percent of applications tested complied with the OWASP Top Ten list and 30 percent with the CWE/SANS Top 25 industry standards
9. **With 62% of applications failing to reach compliance on first submission, procedures for managing non-compliant applications are an important aspect of an enterprise's security policy**

10. 10.11% of vendors resubmitted new versions of applications for testing but are still out of compliance with enterprise policies
11. Structured Testing Programs Promote Higher Participation
a) Enterprises that relied on an ad-hoc approach when requesting application security testing averaged four participating vendors, whereas enterprises with a structured approach had much higher levels of success, averaging participation from 38 vendors.

b) Enterprises with structured programs enabled more vendors to achieve compliance quickly, with 45 percent of vendor applications becoming compliant within one week.

c) By contrast, enterprises with an ad hoc program only saw 28 percent of third-party applications achieve compliance within one week

"Today, every organization is an extended enterprise, with third-party software a fundamental layer in the software supply chain," said Wendy Nather , research director, 451 Research. "It's critical that organizations develop security policies when purchasing software from outside vendors because of the risks inherent in using third-party applications, yet few are actually demanding security compliance of their suppliers."

II. Research Methodology:

This Study of Enterprise Testing of the Software Supply Chain captures data collected from 939 application versions (across 564 distinct applications) submitted to the Veracode Platform during an 18 month time period from January 201 to June 2012. The data comes from actual security analysis of web and non-web applications across industry verticals, languages and platforms, and represents multiple security testing methodologies on a wide range of application types and programming languages.

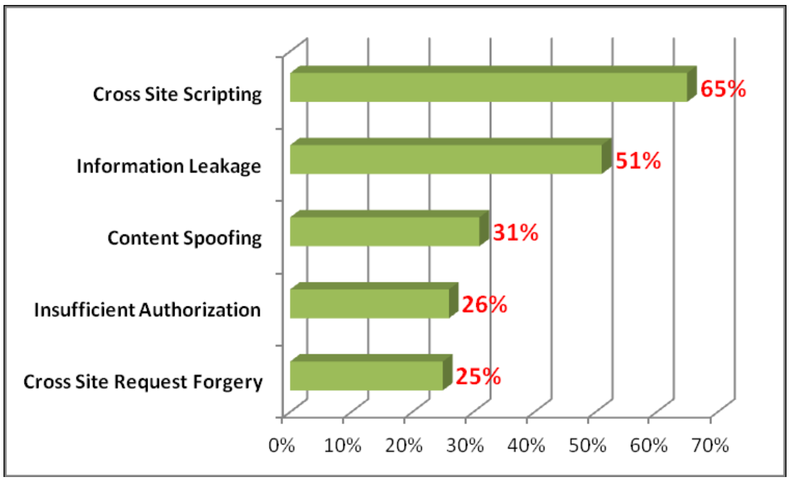
The focus of this report is to assess the security of software purchased from vendors, as well as participating in vendor application security testing programs, and how different programs impact vendor compliance with application security policies.

Web Application Vulnerability Statistics of 2012 by IVIZ:

Key Findings:

- **99%** of the Apps tested had at least 1 vulnerability
- **82%** of the web application had at least 1 High/Critical Vulnerability
- **90%** of hacking incidents never gets known to public
- Very low correlation between Security and Compliance (Correlation Coefficient: **0.2**)
- Average number of vulnerability per website: **35**
- **30%** of the hacked organizations knew the vulnerability (for which they got hacked) beforehand
- **#1** Vulnerability: Cross site scripting (61%)
- **#1** Secure vertical: Banking
- **#1** Vulnerable Vertical: Retail

Top 5 Application Flaws:



Research Methodology:

Application security Data Collection

•300+ Customers

•5,000 + Application Security Tests

•25% Apps from Asia, 40% Apps from USA and 25% from Europe

List of the Scanning tools available:

The following list of products and tools provide web application security scanner functionality. Note that the tools on this list are not being endorsed by the Web Application Security Consortium - any tool that provides web application security scanning functionality will be listed here.

Commercial Tools

- Acunetix WVS by Acunetix
- AppScan by IBM
- Burp Suite Professional by PortSwigger
- Hailstorm by Cenzic
- N-Stalker by N-Stalker
- Nessus by Tenable Network Security
- NetSparker by Mavituna Security
- NeXpose by Rapid7
- NTOSpider by NTObjectives
- ParosPro by MileSCAN Technologies
- Retina Web Security Scanner by eEye Digital Security
- WebApp360 by nCircle
- WebInspect by HP
- WebKing by Parasoft
- Websecurify by GNUCITIZEN

Software-as-a-Service Providers

- AppScan OnDemand by IBM
- ClickToSecure by Cenzic
- QualysGuard Web Application Scanning by Qualys
- Sentinel by WhiteHat
- Veracode Web Application Security by Veracode
- VUPEN Web Application Security Scanner by VUPEN Security
- WebInspect by HP
- WebScanService by Elanize KG

Free / Open Source Tools

- Arachni by Tasos Laskos
- Grabber by Romain Gaucher
- Grendel-Scan by David Byrne and Eric Duprey
- Paros by Chinotec
  - Andiparos
  - Zed Attack Proxy
- Powerfuzzer by Marcin Kozlowski
- SecurityQA Toolbar by iSEC Partners
- Skipfish by Michal Zalewski
- W3AF by Andres Riancho
- Wapiti by Nicolas Surribas
- Watcher by Casaba Security
- WATOBO by siberas
- Websecurify by GNUCITIZEN
- Zero Day Scan

III. Conclusion:

A major contribution of this study is that, it provides details about the importance of application security in secure development, an insight in to the secure development life cycle supported by recently found vulnerabilities/ flaws details with implementation suggestions, survey conducted by different vendors about secure development and currently available tools of scanning which helps in identifying the flaws & ensure secure development.