Java Application Vulnerabilities:

What They Are and How to Fix Them

WRITTEN BY **RYAN O'LEARY,** VICE PRESIDENT, WHITEHAT SECURITY

CONTENTS

Half of all enterprise applications written in the last 15 years have been written in Java, making them nearly ubiquitous in the enterprise. Unfortunately, this means that Java applications are also among the applications hackers most frequently target and attack.

Java can be subject to attack based on general vulnerability types, but there are also some vulnerabilities that are specific to the Java platform.

- Vulnerabilities in standard libraries
- Vulnerabilities introduced by coding errors (e.g., improper construction of a query)

Java-specific vulnerabilities include:

- Vulnerabilities in Java libraries
- Vulnerabilities in the Java sandboxing mechanism, which can allow an attacker to circumvent the restrictions the security manager has established

Developing secure Java-based applications, free from any of the above vulnerabilities, is the best way to ensure that applications are robust and immune to security threats. Incorporating security into the development workflow helps developers avoid creating vulnerabilities; correcting a potential vulnerability during development is exponentially cheaper in both time and resources than correcting a vulnerability that has been implemented in production.

This Refcard is intended to help Java developers understand the most common Java vulnerabilities and how to fix them early in the development process.

TOP JAVA VULNERABILITIES

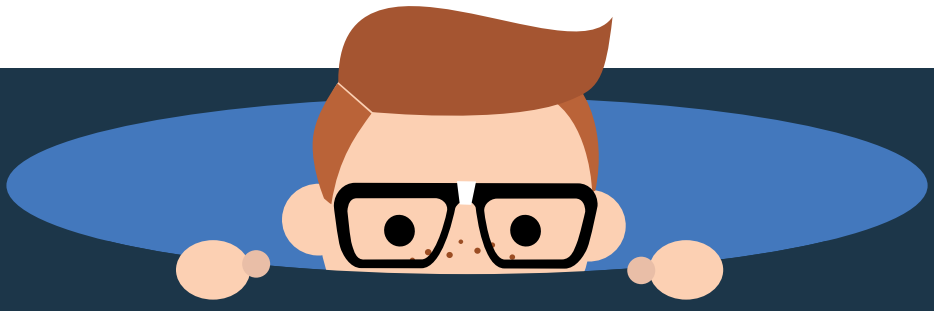
Below are the most common, prevalent, and significant Java vulnerabilities. This list is compiled from the vulnerabilities found in Java applications as reported in the WhiteHat Security Application Security Statistics Report for 2017. For each vulnerability type, you will find a description of how and where it occurs, examples of how to fix it, and other general information about the vulnerability.

IBM Security

Stop hiding from your security officer.

#CoverYourApps with IBM Security's leading application security solution.
Introduce a strong, security-first mindset into your DevOps culture, without disrupting agile practices that your organization relies on to produce quality, on-time software.
Start your free trial today!

IBM Application Security on Cloud enhances web and mobile application security, improves application security program management and strengthens regulatory compliance. Testing web and mobile applications prior to deployment helps you identify security risks, generate reports and receive fix recommendations.



Access Complimentary Trial Now

IBM Application Security on Cloud

AND THE WINNERS ARE…

The most common code vulnerability evident in static application security testing (SAST) during the software development process is Unpatched Libraries. Why? Because modern software is largely assembled of separate components, and everybody uses open source libraries today. These libraries offer readily available options, but are not very secure.

The same applies to the second most prevalent error: Application Misconfiguration. Many software components such as embeddable debug and QA features worry little about security. Developers will enable them by default, creating configuration weaknesses that attackers can exploit. These features may provide a means to bypass authentication methods and gain access to sensitive information, perhaps with elevated privileges.

The answer? Software Composition Analysis (SCA), which is critical when it comes to securing third-party or open source code. SCA looks deeply at the source code of proprietary, open source, and commercial code to identify and inventory all vulnerabilities present.

Critical software errors such as SQL Injection must be fixed during development to reduce exposure in production.

VULNERABILITY RISK RATINGS

Vulnerabilities are rated on five levels of risk – Critical, High, Medium, Low, and Note. Critical and high risk vulnerabilities taken together are referred to as "serious" vulnerabilities. There are three classes of vulnerabilities found to be critical across thousands of applications analyzed in development: Insufficient Transport Layer Protection (ITLP), SQL Injection (SQLi) and Unpatched Libraries.

Insufficient Transport Layer Protection (ITLP)	This is the most critical of them, at 94 percent of vulnerabilities. This is a class used to describe errors such as weak ciphers, certificate misconfiguration or known vulnerable protocols. While ITLP is highly likely to be found by DAST scans, it is rarely a critical error because many organizations are mitigating it via a Web Application Firewall (WAF). A better fix is for developers to properly configure protocols, ciphers and certificates to make them safe. This should be a "must do" during the development process so this error does not make it into live applications.
SQL Injection (SQLi)	This also suffers a very high serious-to-critical ratio, at 81 percent. Both SAST and dynamic application security testing (DAST) detect SQLi criticality equally well, but unlike ITLP it cannot be fought with a firewall. While SQLi attacks are not easily mitigated, they are easily preventable, highlighting the importance of remediating these issues in development.
Unpatched Libraries	These are not only the vulnerabilities most frequently found by SAST, but also one third of these are critical vulnerabilities. Open source components such as libraries must be fixed in development. The best method is to include Software Composition Analysis testing which examines the security of all source code, including components. Certain vulnerabilities can be mitigated in production, while others like SQLi must always be remediated in development. A variety of software security testing regimens routinely performed across the SDLC is the best application security approach. Platform solutions provide this level of visibility and control, leaving organizations with enough intelligence to understand how best to fix any software error… for the least cost.

UNPATCHED LIBRARIES

Critical Risk: OWASP A9: Stat Report Rank 1

DESCRIPTION

Unpatched libraries can introduce critical risks to your application. Utilizing such a library can introduce vulnerabilities, potentially bypassing security controls that are in place elsewhere. Attackers can take advantage of several well-known information sources, such as the National Vulnerability Database, US-CERT, CVE Database, and more to identify these potential vulnerabilities and can use them to introduce almost any weakness.

SOLUTION

Be sure to keep components up-to-date and patched. Monitor for reported vulnerabilities so that prompt action may be taken. Use a dependency manager (e.g. Maven) to declare a minimum

version of any dependencies that are declared multiple times or are declared transitively (dependency of a dependency). Prevent systems and services from leaking unnecessary information, such as version information. If, for some reason, a vulnerable library cannot be patched or replaced, ensure that compensating controls are in place, such as a properly configured network firewall, IDS/IPS, or application firewall.

Before selecting components, always perform research into known vulnerabilities. Store these findings in a central repository, include lists of libraries known to be vulnerable, and ensure that all development teams have access to this information so that vulnerable libraries can be dealt with immediately, so these components are less likely to be used by accident. Be sure to consider the actual impact of any vulnerabilities identified. In some cases, the risk may be much higher than normal and in other cases, the vulnerability may not apply to how the component will be used.

Establish a company-wide governance policy for selecting, testing, and approving components for use by development teams. When components are approved for use, store them in a central repository and share them with other development teams throughout the organization. It is best to have a single solution rather than several solutions to the same problem. To further improve consistency and security assurance, version and patch levels should remain consistent throughout the organization.

- Dependency Management tools (e.g. Maven)
- WhiteHat Sentinel Source provides information on libraries and frameworks for covered assets.

ADDITIONAL RESOURCES

maven.apache.org/guides/introduction/introduction-todependency-mechanism.html#Dependency_Management

OWASP TOP 10 VULNERABILITIES

A1	Injection
A2	Broken Authentication and Session Management (XSS)
A3	Cross Site Scripting (XSS)
A4	Insecure Direct Object References
A5	Security Misconfiguration
A6	Sensitive Data Exposure
A7	Missing Function Level Access Control
A8	Cross Site Request Forgery (CSRF)
A9	Using Components with Known Vulnerabilities
A10	Unvalidated Redirects and Forwards

APPLICATION MISCONFIGURATION: EXPOSED SERVELET

Critical Risk: OWASP A5: Stat Report Rank 2

DESCRIPTION

This Axis application is configured to deploy an administration interface. This interface can be viewed without use of normal Authentication/Access restrictions. Malicious users could use this interface to get access to unintended server functionality. Cases where the application is “internal only” have a reduced likelihood to reflect the need for internal network access. However, exposing unauthenticated administrative functionality even to the internal network is not secure, and should still be considered a vulnerability with some level of risk.

SOLUTION

Remove the highlighted snippet from the production web.xml. Since the neither the AdminServlet and SOAPMonitorService support acceptable authentication schemes, disabling these servlets is the only secure option.

ADDITIONAL RESOURCES

axis.apache.org/axis/java/security.html

APPLICATION MISCONFIGURATION: EXCESSIVE PERMISSIONS

Low Risk: OWASP A5: Stat Report Rank 2

DESCRIPTION

An application may use custom permissions that can then allow a separate application to access hardware level functionality through its API. These separate applications can bypass the normal prompting procedures for use of sensitive functionalityby using the API.

SOLUTION

Applications should only request the minimum permissions needed for stated application functionality. Do not request any unnecessary permissions. Any unused permissions should not be requested. Future application updates should prompt the user to revoke unneeded permissions.

APPLICATION MISCONFIGURATION: GLOBAL ERROR HANDLING DISABLED

Medium Risk: OWASP A5: Stat Report Rank 2

DESCRIPTION

Disabling a global error handling mechanism increases the risk

that verbose implementation details will be revealed to attackers through a stack trace.

SOLUTION

To minimize the risk of disclosing sensitive implementation details through error messages, ensure the application deployment descriptor declares an error-page declaration that catches all uncaught exceptions thrown by the application.

EXAMPLES

The web.xml should define error handling elements such as:

```
<error-page>
<error-code>500</error-code>
<location>/path/to/default_500.jsp</location>
</error-page>
<error-page>
<exception-type>java.io.IOException</exception-type>
<location>/path/to/default_exception_handler.jsp</
location>
</error-page>
```

CROSS-SITE SCRIPTING ("XSS")

High Risk: OWASP A3: Stat Report Rank 3

DESCRIPTION

Cross-site scripting (sometimes referred to as "XSS") vulnerabilities occur when an attacker embeds malicious clientside script or HTML in a form or query variables submitted to a site via a web interface, sending the malicious content to an end-user. If this content is submitted by one user (the attacker), stored in the database, and subsequently rendered to a different user (the victim), a Persisted Cross Site Scripting Attack occurs. A variation of this attack, known as Reflected Cross Site Scripting, occurs when an attacker entices a victim into submitting the tainted data themselves, via an email or a link on an attackercontrolle website.

An attacker can use XSS to send malicious script or other content to an unsuspecting user. Neither the end user nor their browser knows that the content was not actually generated by the trusted website. The malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used by the site. These scripts can even rewrite the content of the HTML page. This can result in phishing attacks, identity theft, website defacement, denial-of-service, and other attacks.

SOLUTION

The most reliable means of thwarting most types of XSS attacks is to HTML-encode or URL-encode all output data, regardless of the data's source. This ensures that tainted data can't affect the output from any source, including user input and information shared with other applications or originating from third-party sources. Consistently encoding all output data also makes the application much easier to audit, since it eliminates the need to perform time consuming (and expensive) data flow analysis. It is important to note that there are different output contexts which encoding functionality must handle, including HTML, HTML attributes, URLs, CSS, and JavaScript. A single encoding approach will not necessarily mitigate XSS in every context.

If output encoding isn't practicable, the next most effective approach is to carefully filter all input data against a white-list of allowed characters. This approach does have the advantage that it can be performed externally without modifying application source code. Data retrieved from third-parties or shared with other applications should be filtered along with user input data. The white-list should only include characters which may be a legitimate part of user input. The following characters are especially useful for conducting XSS attacks and should be considered in any encoding or filtering scheme: < > " ' ; & ? One or more of these characters, such as the single quote, may be required by the application. If it is impracticable to remove one or more dangerous characters as part of an input filtering scheme, they must be carefully handled. Such characters could, for example, be encoded on input, and stored encoded in the database. Ideally, the application should perform careful input validation and use output encoding to guard against XSS and other injection attacks.

ADDITIONAL RESOURCES

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

https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

CRYPTOGRAPHY: IMPROPER PSEUDO-RANDOM NUMBER GENERATOR USAGE

Medium Risk: OWASP NA: Stat Report Rank 4

DESCRIPTION

Insufficient randomness results when software generates predictable values when unpredictability is required. When a security mechanism relies on random, unpredictable values to restrict access to a sensitive resource, such as an initialization vector (IV), a seed for generating a cryptographic key, or a session ID, then use of insufficiently random numbers may allow an attacker to access the resource by guessing the value. The potential consequences of using insufficiently random numbers are data theft or modification, account or system compromise, and loss of accountability – i.e., non-repudation.

SOLUTION

When using random numbers in a security context, use cryptographically secure pseudo-random number generators (CSPRNG).

EXAMPLES

```
byte[] randomBytes = new byte[8];
SecureRandom random = new SecureRandom();
random.nextBytes(randomBytes);
```

APPLICATION MISCONFIGURATION: DEBUG

Medium Risk: OWASP A5: Stat Report Rank 5

DESCRIPTION

Application errors commonly occur during normal operation, particularly when the application is misused, even unintentionally. If debugging is enabled, then, when occurs occur, the application may provide inside information to end-users who should not have access to it and who may use it to attack the application. Error messages displayed to an end user could include server information, a detailed exception message, a stack trace, or even the actual source code of the page where the error occurred. This information could be used to help formulate an attack.

If the application provides a switch to enable debug mode in production, attackers could guess or learn of this parameter and take advantage of any additional information the application may provide. Custom debug mode implementations have even been observed to bypass authentication or assign administratorlevel permissions for testing purposes.

SOLUTION

Debug mode should be disabled in production. In addition to any debug mode provided by the programming language, developers may implement their own custom debug mode. Custom debug options should be stored in application configuration files rather than source code, but it may be necessary to search the code base to verify there are no hidden debug options.

Production code should normally not be capable of entering debug mode or producing debug messages. However, if this capability is necessary, debug mode should be triggered by editing a file or configuration option on the server. In particular, debug should not be enabled by an option in the application itself. For example, it should not be possible to pass in a URL parameter to trigger debug mode, such as the following: www.website.com?debug=true. Regardless how obscure the parameter may be, it is never a secure option.

Frameworks and components used by the application may have their own debug options as well. It is important that debug options are disabled throughout the application before the application is deployed.

EXAMPLES

There are many instances were a Debug mode may exist within a Java application, and this varies depending on the container. Here is one example of debug_mode disabled for the jsp servlet:

```
<servlet>
<servlet-name>jsp</servlet-name>
<servlet-class>oracle.jsp.runtimev2.JspServlet</servletclass>
<init-param>
<param-name>debug_mode</param-name>
<param-value>false</param-value>
</init-param>
```

Since developers may have implemented their own custom debug mode, be sure to inspect configuration files and search the code base for things like:

```
DEBUG MODE
debug = true
debug = 1
debug
```

DISCLOSURE: CLEARTEXT PASSWORD

Medium Risk: OWASP A2: Stat Report Rank 6

DESCRIPTION

If an application contains one or more hardcoded passwords within the source code, an attacker with access to the source code or compiled binaries can extract the credentials in an attempt to access the corresponding services. Obscuring passwords using encoding, such as Base-64, is not sufficient. Storing passwords in cleartext (e.g. in an application's properties or configuration file) can result in account or system compromise. This exposes the password to any personnel with access to the application's configuration files – developers, architects, testers, auditors, and development managers. Because it is possible that others may have access to a user's password, the owner of an account can no longer be presumed to be the only person able to login to the account.

SOLUTION

System passwords should be encrypted, or the configuration file they are contained within should be encrypted, whenever possible. Credentials should be encrypted with a key and stored on disk in a non-web-accessible directory, read-only accessible to the user running the web application (webserver). The key should be stored in a separate non-web-accessible location that is also read-only to the user running the web application. The application can then read the key (from a known static location), read the encrypted credentials, decode, and use them. The encryption key should be rotated on a 30- to 90-day basis. User passwords should be stored using a strong one-way hashing algorithm, such as SHA-256. A cryptographic salt should be added to each password before it is hashed. The salt should be at least, 64-bits in length and should be random or unique to each user.

EXAMPLES

There are many approaches and libraries available for encrypting/ecrypting data in Java. Many common approaches are less than secure. Java developers often encode system passwords in Base-64 or encrypt them with DES - neither approach is secure, especially encoding.

The following code can be used to encrypt/decrypt using a secure algorithm, AES:

```
public static String encrypt(String value, File keyFile)
 throws GeneralSecurityException, IOException {
 if (!keyFile.exists()) {
 KeyGenerator keyGen = KeyGenerator.
getInstance(CryptoUtils.AES);
 keyGen.init(128);
 SecretKey sk = keyGen.generateKey();
 FileWriter fw = new FileWriter(keyFile);
 fw.write(byteArrayToHexString(sk.getEncoded()));
 fw.flush();
 fw.close();
 }
 SecretKeySpec sks = getSecretKeySpec(keyFile);
 Cipher cipher = Cipher.getInstance(CryptoUtils.AES);
 cipher.init(Cipher.ENCRYPT_MODE, sks, cipher.
getParameters());
 byte[] encrypted = cipher.doFinal(value.getBytes());
 return byteArrayToHexString(encrypted);
}
```

INJECTION: UNKNOWN INTERPRETER

Medium Risk: OWASP A5: Stat Report Rank 7

DESCRIPTION

The application makes use of untrusted data in conjunction with the creation and or use of an interpreter. Untrusted data is retrieved from the attacker and utilized as an argument to a dangerous interpreter access method. Failure to properly validate or encode data utilized by an interpreter increases the risk of injection attacks. Such injection typically results in the attacker's ability to execute arbitrary code in the context of the program consuming the interpreter results.

SOLUTION

Define and enforce a strict set of criteria defining what the application will accept as valid input, and contextually encode all untrusted data passed to the interpreter prior to execution.

DENIAL OF SERVICE (DOS): READLINE

Medium Risk: OWASP NA: Stat Report Rank 8

DESCRIPTION

The java.io.BufferedReader readLine() method can be used to read data from a socket or file; however, readLine() reads data until it encounters a newline or carriage return character in the data. If neither of these characters are found, readLine() will continue reading data indefinitely. If an attacker has any control over the source being read, he or she can inject data that does not have these characters and cause a denial of service on the system. Even if the number of lines to be read is limited, an attacker can supply a large file with no newline characters and cause an OutOfMemoryError exception.

SOLUTION

OWASP's Enterprise Security API (owasp.org/index.php/Category:OWASP_Enterprise_Security_API) provides a safer alternative to readLine() called SafeReadLine(). This method reads from an input stream until end-of-line or the maximum number of characters is reached, effectively mitigating this risk.

Another solution is to override both BufferedReader and the readLine() method and implement a limit for the maximum number of characters that can be read. In the absence of a more secure method, avoid taking input from the client whenever possible and ensure data being read is trusted.

EXAMPLES

OWASP ESAPI's safeReadLine() can be used to safely read untrusted data as follows.

```
ByteArrayInputStream s = new
ByteArrayInputStream("testinput".getBytes());
IValidator instance = ESAPI.validator();
try {
 String u = instance.safeReadLine(s, 20);
} catch (ValidationException e) {
 // Handle exception
}
```

URL REDIRECTOR ABUSE

Medium Risk: OWASP A10: Stat Report Rank 9

DESCRIPTION

Applications frequently redirect users to other pages using stored URLs. Sometimes the target page is specified in an untrusted parameter, allowing attackers to choose the destination page or location. Such redirects may improperly leverage the trust the user has in the ulnerable website.

SOLUTION

If untrusted data becomes part of a redirect URL, ensure that the supplied value has been properly validated and was part of a legal and authorized request from the user. It is recommended that legal redirect destinations be driven by a "destination id" that is mapped to the actual redirect destination server-side, rather than the actual URL or portion of the URL originating from

the user request. Lookup-maps or access controls tables are best for this purpose.

INSUFFICIENT SESSION EXPIRATION

Medium Risk: OWASP A2: Stat Report Rank 10

DESCRIPTION

PCI Data Security Standards Version 3, Section 8.1.8 specifies a maximum session timeout of 15 minutes for critical components of an application: "If a session has been idle for more than 15 minutes, require the user to re-authenticate to re-activate then terminal or session."

User sessions with long or no inactivity timeouts may help attackers replay attacks or hijack sessions. Social engineering attacks are also more likely to succeed with a longer time-out. An attacker has a greater opportunity to gain physical access to a user's machine if a user does not close the application.

If the session timeout is not specified in a web application's configuration, the default value will be used, which is often 24, 30, or 60 minutes, depending on the web server, version, and its configuration.

SOLUTION

In general, idle user sessions should timeout within 15-20 minutes, or less for sensitive applications. Consider disabling "sliding expiration" if the configuration option exists. If it is necessary to enable this option, consider implementing a hard session timeout in addition to the sliding timeout. When sessions timeout, the application should invalidate the session, removing session data as well as any cookies and authentication tokens.

EXAMPLE

The session timeout can be configured at the server-level in the default web.xml or for each web application individually. The following code should be included in the application's web.xml file:

```
<session-config>
<session-timeout>15</session-timeout>
</session-config>
```

If WebLogic is being used, the session timeout should also be specified in the weblogic.xml file. The following code sets the timeout to 15 minutes:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<weblogic-web-app xmlns="http://www.bea.com/ns/
weblogic/90">
<session-descriptor>
<timeout-secs>900</timeout-secs>
</session-descriptor>
</weblogic-web-app>
```

MISSING ACCESS STRATEGY

High Risk: OWASP A5: Stat Report Rank 13

DESCRIPTION

This application is not utilizing an access control strategy for one or more components. Failure to utilize access control can lead to exposure of sensitive functionality to unintended users. Malicious users seek out this type of functionality to cause harm to users of the application, or the application itself. In Websphere, if you enable servlets by class name, then this is performing the same act as Android in that it allows you to invoke by the class. If the following snippet exists or the variable is not declared, this allows you to invoke servlets without any permissions:

```
enable-serving-servlets-by-class-name value="true"
```

SOLUTION

Utilize an access control strategy for all components of the application where sensitive functionality may reside. Prevent servlets from serving by classname by adding the following line:

```
enable-serving-servlets-by-class-name value="false"
```

INSUFFICIENT TRANSPORT LAYER PROTECTION

DESCRIPTION

Applications frequently fail to encrypt network traffic when it is necessary to protect sensitive communications. Encryption (usually TLS) must be used for all authenticated connections, especially Internet-accessible web pages. Backend connections should be encrypted as well. Otherwise, the application will expose an authentication or session token to malicious actors on the same network as the application host. These backend connections may represent a lower likelihood of exploitation than a connection over the external internet; however, in the case of exploitation they can result in compromise of user accounts or worse.

Encryption should be used whenever sensitive data, such as credit card or health information, is transmitted. Applications that fall back to plaintext or are otherwise forced out of an encrypting mode can be abused by attackers.

SOLUTION

Ensure the application has a security constraint that defines a confidentiality and integrity-based secure transport guarantee. This will ensure that all data is sent in a manner that guarantees it cannot be observed or changed during transmission. If TLS must be terminated at a load balancer, web application firewall, or other inline host, it should re-encrypt the data in transit to the target host(s).

CONCLUSION

All organizations must implement application security procedures early in their SDLC. It has been proven to be faster and less expensive to catch and fix flaws earlier rather than later. Routine security testing in development makes resulting production applications stronger. Organizations that integrate multiple kinds of testing regimens (e.g., DAST, SAST, mobile, etc.) directly with their SDLC see the best results. Today's application security platforms extend visibility and control even further with Software Composition Analysis, API testing, developer training, and other services.

From WannaCry to the September 2017 Equifax breach, we are reminded almost daily that that the failure to address application vulnerabilities can lead to terrible – but avoidable – attacks. Adoption of secure DevOps, or DevSecOps, is imperative for true security in the age of digital business.

Written by **Ryan O'Leary,** Vice President, WhiteHat Security

Ryan is Vice President of the Threat Research Center at WhiteHat Security, the specialized team of web application security experts. Ryan joined WhiteHat Security as an ethical hacker in 2007 and has since developed a breadth of experience finding and exploiting web application vulnerabilities and configuring automated tools for testing. Ryan manages a team of over 150 security engineers, and is also responsible for overseeing the delivery of the WhiteHat Sentinel Application Security Platform, which services over 10,000 customer websites. Under Ryan's leadership, the team has built a one-of-a-kind database that combines details of many millions of million vulnerability patterns with proprietary algorithms to assess the threat level.

As one of the world's leading experts on application security, Ryan is frequently turned to by the media for commentary on some of the most pressing cyber security issues.

DZone, Inc.
150 Preston Executive Dr. Cary, NC 27513
888.678.0399 919.678.0300