VERACODE

Veracode Detailed Report

Application Security Report
As of 14 Nov 2017

Prepared for:	Tufts University
Prepared on:	April 5, 2018
Application:	Facebook Follower Counter Bot
Industry:	Not Specified
Business Criticality:	BC3 (Medium)
Required Analysis:	Static
Type(s) of Analysis Conducted:	Static
Scope of Static Scan:	2 of 25 Modules Analyzed

Inside This Report

Veracode Detailed Report

Application Security Report
As of 14 Nov 2017

Veracode Level: VL1
Rated: Nov 14, 2017

Application:	Facebook Follower Counter Bot	Business Criticality:	Medium
Target Level:	VL3	Published Rating:	B

Scans Included in Report

Static Scan	Dynamic Scan	Manual Scan
Facebook Bot Scan 1 Score: 74 Completed: 11/14/17	Not Included in Report	Not Included in Report

Executive Summary

This report contains a summary of the security flaws identified in the application using automated static, automated dynamic and/or manual security analysis techniques. This is useful for understanding the overall security quality of an individual application or for comparisons between applications.

Application Business Criticality: BC3 (Medium)

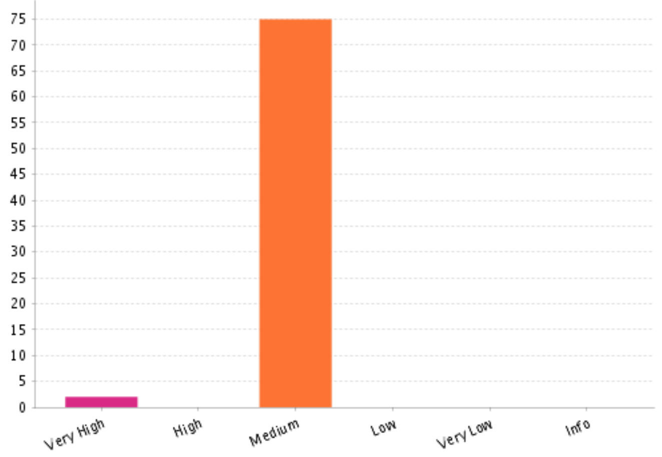
Impacts:Operational Risk (Low), Financial Loss (Medium)

An application's business criticality is determined by business risk factors such as: reputation damage, financial loss, operational risk, sensitive information disclosure, personal safety, and legal violations. The Veracode Level and required assessment techniques are selected based on the policy assigned to the application.

Analyses Performed vs. Required

	Any	Static	Dynamic	Manual
Performed:		●	○	○
Required:	○	●	○	○

Summary of Flaws Found by Severity



Action Items:

Veracode recommends the following approaches ranging from the most basic to the strong security measures that a vendor can undertake to increase the overall security level of the application.

Required Analysis

→ Your policy requires periodic Static Scan and you are overdue. Please submit your application for Static Scan and remediate the required detected flaws to conform to your assigned policy.

Flaws To Fix By Expires Date

A grace period is specified for any flaw that violates the rules contained in your policy. These include CWE, Rollup Category, Issue Severity, Industry Standards as well as any flaws the prevent an application from achieving a minimum Veracode Level and/or score. To maintain policy compliance you must fix these flaws and resubmit your application for scanning before the grace period expires. The detailed flaw listing will badge the flaws that must be fixed and show the fix by date as well.

→ The grace period has expired [11/14/17] for 2 flaws that were found in your Static Scan.

Flaw Severities

- High severity flaws and above must be fixed for policy compliance.

Longer Timeframe (6 - 12 months)

- Certify that software engineers have been trained on application security principles and practices.

Scope of Static Scan

The following modules were included in the static scan because the scan submitter selected them as entry points, which are modules that accept external data.

Engine Version: 117385

The following modules were included in the application scan:

Module Name	Compiler	Operating Environment	Engine Version
JS files within followy-master.zip	JAVASCRIPT_5_1	JavaScript	117385
Python files within followy-master.zip	Python		117385

The following modules were not selected for a full scan. Code paths in these modules that are not called from a scanned module are not included in this report.

Module Name	Compiler	Operating Environment	Engine Version
chardetect.exe	MSVC10_X86	Win32	117385
cli-32.exe	MSVC9_X86	Win32	117385
cli-64.exe	MSVC9_X86_64	Win64	117385
cli.exe	MSVC9_X86	Win32	117385
easy_install-3.6.exe	MSVC10_X86	Win32	117385
easy_install.exe	MSVC10_X86	Win32	117385
flask.exe	MSVC10_X86	Win32	117385
gui-32.exe	MSVC9_X86	Win32	117385
gui-64.exe	MSVC9_X86_64	Win64	117385
gui.exe	MSVC9_X86	Win32	117385
gunicorn.exe	MSVC10_X86	Win32	117385
gunicorn_paster.exe	MSVC10_X86	Win32	117385
pip.exe	MSVC10_X86	Win32	117385
pip3.6.exe	MSVC10_X86	Win32	117385
pip3.exe	MSVC10_X86	Win32	117385
python.exe	MSVC14_X86	Win32	117385
python36.dll	MSVC14_X86	Win32	117385
pythonw.exe	MSVC14_X86	Win32	117385
t32.exe	MSVC10_X86	Win32	117385
t64.exe	MSVC10_X86_64	Win64	117385
w32.exe	MSVC10_X86	Win32	117385
w64.exe	MSVC10_X86_64	Win64	117385
wheel.exe	MSVC10_X86	Win32	117385

Flaw Types by Severity and Category

	Static Scan Security Quality Score = 74		
Very High	**2**		
Code Injection	2		
High	**0**		
Medium	**75**		
CRLF Injection	1		
Cross-Site Scripting	3		
Cryptographic Issues	54		
Directory Traversal	14		
Server Configuration	3		
Low	**0**		
Very Low	**0**		
Informational	**0**		
Total	**77**		

Policy Evaluation

Policy Name: Veracode Recommended Medium

Revision: 1

Policy Status: Did Not Pass

Description

Veracode provides default policies to make it easier for organizations to begin measuring their applications against policies. Veracode Recommended Policies are available for customers as an option when they are ready to move beyond the initial bar set by the Veracode Transitional Policies. The policies are based on the Veracode Level definitions.

Rules

Rule type	Requirement	Findings	Status
Minimum Veracode Level	VL3	VL1	Did not pass
(VL3) Min Analysis Score	70	74	Passed
(VL3) Max Severity	High	Flaws found: 2	Did not pass

Scan Requirements

Scan Type	Frequency	Last performed	Status
Static	Quarterly	11/14/17	Did not pass

Remediation

Flaw Severity	Grace Period	Flaws Exceeding	Status
Very High	0 days	2	Did not pass
High	0 days	0	Passed
Medium	0 days	0	Passed
Low	0 days	0	Passed
Very Low	0 days	0	Passed
Informational	0 days	0	Passed

Type	Grace Period	Exceeding	Status
Min Analysis Score	0 days	0	Passed

Findings & Recommendations

Detailed Flaws by Severity

Very High (2 flaws)

Fix Required by Policy

→ Code Injection(2 flaws)

Description

Code injection is the process of injecting untrusted input into an application that dynamically evalutes and executes the input as code. Common examples of code injection include Remote File Includes and Eval Injection into applications implemented in an interpreted language such as PHP.

Recommendations

Do not allow untrusted input to be evaluated or otherwise interpreted as code.

Associated Flaws by CWE ID:

→ Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection') (CWE ID 95)(2 flaws)

Description

The software allows untrusted input to be fed directly into a function (e.g. "eval") that dynamically evaluates and executes the input as code, usually in the same interpreted language that the product uses.

Effort to Fix: 3 - Complex implementation error. Fix is approx. 51-500 lines of code. Up to 5 days to fix.

Recommendations

Validate all untrusted input to ensure that it conforms to the expected format, using centralized data validation routines when possible. In general, avoid executing code derived from untrusted input.

Instances found via Static Scan

	Flaw Id	Module #	Class #	Module	Location	Fix By
✖	25	33	-	Python files within followy-master.zip	/followy-master/.../launch.py 30	11/14/17
✖	68	33	-	Python files within followy-master.zip	/followy-master/.../launch.py 31	11/14/17

High (0 flaws)

No flaws of this type were found

Medium (75 flaws)

→ CRLF Injection(1 flaw)

Description

The acronym CRLF stands for "Carriage Return, Line Feed" and refers to the sequence of characters used to denote the end of a line of text. CRLF injection vulnerabilities occur when data enters an application from an untrusted source and is not properly validated before being used. For example, if an attacker is able to inject a CRLF into a log file, he could append falsified log entries, thereby misleading administrators or cover traces of the attack. If an attacker is able to inject CRLFs into an HTTP response header, he can use this ability to carry out other attacks such as cache poisoning. CRLF vulnerabilities primarily affect data integrity.

Recommendations

Apply robust input filtering for all user-supplied data, using centralized data validation routines when possible. Use output filters to sanitize all output derived from user-supplied input, replacing non-alphanumeric characters with their HTML entity equivalents.

Associated Flaws by CWE ID:

→ Improper Output Neutralization for Logs (CWE ID 117)(1 flaw)

Description

A function call could result in a log forging attack. Writing untrusted data into a log file allows an attacker to forge log entries or inject malicious content into log files. Corrupted log files can be used to cover an attacker's tracks or as a delivery mechanism for an attack on a log viewing or processing utility. For example, if a web administrator uses a browser-based utility to review logs, a cross-site scripting attack might be possible.

Effort to Fix: 2 - Implementation error. Fix is approx. 6-50 lines of code. 1 day to fix.

Recommendations

Avoid directly embedding user input in log files when possible. Sanitize untrusted data used to construct log entries by using a safe logging mechanism such as the OWASP ESAPI Logger, which will automatically remove unexpected carriage returns and line feeds and can be configured to use HTML entity encoding for non-alphanumeric data. Only write custom blacklisting code when absolutely necessary. Always validate untrusted input to ensure that it conforms to the expected format, using centralized data validation routines when possible.

Instances found via Static Scan

Flaw Id	Module #	Class #	Module	Location	Fix By
1	19	-	JS files within followy-master.zip	/.../debug/shared/debugger.js 118	

→ Cross-Site Scripting(3 flaws)

Description

Cross-site scripting (XSS) attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed occur whenever a web application uses untrusted data in the output it generates without validating or encoding it. XSS vulnerabilities are commonly exploited to steal or manipulate cookies, modify presentation of content, and compromise sensitive information, with new attack vectors being discovered on a regular basis. XSS is also commonly referred to as HTML injection.

XSS vulnerabilities can be either persistent or transient (often referred to as stored and reflected, respectively). In a persistent XSS vulnerability, the injected code is stored by the application, for example within a blog comment or message board. The attack occurs whenever a victim views the page containing the malicious script. In a transient XSS vulnerability, the injected code is included directly in the HTTP request. These attacks are often carried out via malicious URLs sent via email or another website and requires the victim to browse to that link. The consequence of an XSS attack to a victim is the same regardless of whether it is persistent or transient; however, persistent XSS vulnerabilities are likely to affect a greater number of victims due to its delivery mechanism.

Recommendations

Several techniques can be used to prevent XSS attacks. These techniques complement each other and address security at different points in the application. Using multiple techniques provides defense-in-depth and minimizes the likelihood of a XSS vulnerability.

* Use output filtering to sanitize all output generated from user-supplied input, selecting the appropriate method of encoding based on the use case of the untrusted data. For example, if the data is being written to the body of an HTML page, use HTML entity encoding. However, if the data is being used to construct generated Javascript or if it is consumed by client-side methods that may interpret it as code (a common technique in Web 2.0 applications), additional restrictions may be necessary beyond simple HTML encoding.
* Validate user-supplied input using positive filters (white lists) to ensure that it conforms to the expected format, using centralized data validation routines when possible.
* Do not permit users to include HTML content in posts, notes, or other data that will be displayed by the application. If users are permitted to include HTML tags, then carefully limit access to specific elements or attributes, and use strict validation filters to prevent abuse.

Associated Flaws by CWE ID:

→ Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS) (CWE ID 80)(3 flaws)

Description

This call contains a cross-site scripting (XSS) flaw. The application populates the HTTP response with untrusted input, allowing an attacker to embed malicious content, such as Javascript code, which will be executed in the context of the victim's browser. XSS vulnerabilities are commonly exploited to steal or manipulate cookies, modify presentation of content, and compromise confidential information, with new attack vectors being discovered on a regular basis.

Effort to Fix: 3 - Complex implementation error. Fix is approx. 51-500 lines of code. Up to 5 days to fix.

Recommendations

Use contextual escaping on all untrusted data before using it to construct any portion of an HTTP response. The escaping method should be chosen based on the specific use case of the untrusted data, otherwise it may not protect fully against the attack. For example, if the data is being written to the body of an HTML page, use HTML entity escaping; if the data is being written to an attribute, use attribute escaping; etc. When a web framework provides built-in support for automatic XSS escaping, do not disable it. Both the OWASP Java Encoder library for Java and the Microsoft AntiXSS library provide contextual escaping methods. For more details on contextual escaping, see https://www.owasp.org/index.php/XSS_%%28Cross_Site_Scripting%%29_Prevention_Cheat_Sheet. In addition, as a best practice, always validate untrusted input to ensure that it conforms to the expected format, using centralized data validation routines when possible.

Instances found via Static Scan

Flaw Id	Module #	Class #	Module	Location	Fix By
38	10	-	Python files within followy-master.zip	/followy-master/bot.py 63	
3	19	-	JS files within followy-master.zip	/.../debug/shared/debugger.js 76	
2	19	-	JS files within followy-master.zip	/.../debug/shared/debugger.js 159	

Cryptographic Issues(54 flaws)

Description

Applications commonly use cryptography to implement authentication mechanisms and to ensure the confidentiality and integrity of sensitive data, both in transit and at rest. The proper and accurate implementation of cryptography is extremely critical to its efficacy. Configuration or coding mistakes as well as incorrect assumptions may negate a large degree of the protection it affords, leaving the crypto implementation vulnerable to attack.

Common cryptographic mistakes include, but are not limited to, selecting weak keys or weak cipher modes, unintentionally exposing sensitive cryptographic data, using predictable entropy sources, and mismanaging or hard-coding keys.

Developers often make the dangerous assumption that they can improve security by designing their own cryptographic algorithm; however, one of the basic tenets of cryptography is that any cipher whose effectiveness is reliant on the secrecy of the algorithm is fundamentally flawed.

Recommendations

Select the appropriate type of cryptography for the intended purpose. Avoid proprietary encryption algorithms as they typically rely on "security through obscurity" rather than sound mathematics. Select key sizes appropriate for the data being protected; for high assurance applications, 256-bit symmetric keys and 2048-bit asymmetric keys are sufficient. Follow best practices for key storage, and ensure that plaintext data and key material are not inadvertently exposed.

Associated Flaws by CWE ID:

Insufficient Entropy (CWE ID 331)(25 flaws)

Description

Standard random number generators do not provide a sufficient amount of entropy when used for security purposes. Attackers can brute force the output of pseudorandom number generators such as rand().

Effort to Fix: 2 - Implementation error. Fix is approx. 6-50 lines of code. 1 day to fix.

Recommendations

If this random number is used where security is a concern, such as generating a session identifier or cryptographic key, use a trusted cryptographic random number generator instead.

Instances found via Static Scan

Flaw Id	Module #	Class #	Module	Location	Fix By
47	4	-	Python files within followy-master.zip	/followy-master/.../arbiter.py 612	

Flaw Id	Module #	Class #	Module	Location	Fix By
24	8	-	Python files within followy-master.zip	/followy-master/.../base.py 53	
70	14	-	Python files within followy-master.zip	/.../boto/route53/connection.py 608	
42	15	-	Python files within followy-master.zip	/.../boto/connection.py 922	
48	17	-	Python files within followy-master.zip	/.../dialects/oracle/cx_oracle.py 985	
55	20	-	Python files within followy-master.zip	/followy-master/.../default.py 455	
27	21	-	Python files within followy-master.zip	/followy-master/.../djbec.py 242	
29	21	-	Python files within followy-master.zip	/followy-master/.../djbec.py 260	
59	22	-	Python files within followy-master.zip	/.../command/easy_install.py 437	
34	24	-	Python files within followy-master.zip	/followy-master/.../filters.py 366	
19	30	-	Python files within followy-master.zip	/.../ubuntu/installer.py 41	
52	30	-	Python files within followy-master.zip	/.../ubuntu/installer.py 43	
74	36	-	Python files within followy-master.zip	/.../werkzeug/posixemulation.py 97	
76	37	-	Python files within followy-master.zip	.../resumable_upload_handler.py 564	
9	38	-	Python files within followy-master.zip	/.../pip/_vendor/retrying.py 159	
33	38	-	Python files within followy-master.zip	/.../pip/_vendor/retrying.py 218	
10	39	-	Python files within followy-master.zip	/followy-master/.../serving.py 339	
31	40	-	Python files within followy-master.zip	/.../werkzeug/contrib/sessions.py 77	
28	44	-	Python files within followy-master.zip	/followy-master/.../test.py 48	
45	46	-	Python files within followy-master.zip	/followy-master/.../util.py 75	
6	49	-	Python files within followy-master.zip	/followy-master/.../utils.py 253	
4	49	-	Python files within followy-master.zip	/followy-master/.../utils.py 255	
46	49	-	Python files within followy-master.zip	/followy-master/.../utils.py 263	
72	49	-	Python files within followy-master.zip	/followy-master/.../utils.py 268	
73	50	-	Python files within followy-master.zip	/followy-master/.../boto/utils.py 315	

Use of a Broken or Risky Cryptographic Algorithm (CWE ID 327)(29 flaws)

Description

The use of a broken or risky cryptographic algorithm is an unnecessary risk that may result in the disclosure of sensitive information.

Effort to Fix: 1 - Trivial implementation error. Fix is up to 5 lines of code. One hour or less to fix.

Instances found via Static Scan

Flaw Id	Module #	Class #	Module	Location	Fix By
13	1	-	Python files within followy-master.zip	/.../werkzeug/debug/__init__.py 44	
17	1	-	Python files within followy-master.zip	/.../werkzeug/debug/__init__.py 167	
36	2	-	Python files within followy-master.zip	/.../oauth1/rfc5849/__init__.py 185	
22	5	-	Python files within followy-master.zip	/followy-master/.../auth.py 113	
39	5	-	Python files within followy-master.zip	/followy-master/.../auth.py 119	
64	6	-	Python files within followy-master.zip	/followy-master/.../auth.py 148	
62	5	-	Python files within followy-master.zip	/followy-master/.../auth.py 151	
63	6	-	Python files within followy-master.zip	/followy-master/.../auth.py 154	
11	6	-	Python files within followy-master.zip	/followy-master/.../auth.py 186	
32	9	-	Python files within followy-master.zip	/followy-master/.../bccache.py 168	
40	9	-	Python files within followy-master.zip	/followy-master/.../bccache.py 178	
49	11	-	Python files within followy-master.zip	/followy-master/.../cache.py 165	
18	12	-	Python files within followy-master.zip	/followy-master/.../cache.py 744	
77	16	-	Python files within followy-master.zip	/.../boto/mws/connection.py 57	
54	13	-	Python files within followy-master.zip	/.../boto/sns/connection.py 347	
66	18	-	Python files within followy-master.zip	/.../_vendor/distlib/database.py 991	
71	23	-	Python files within followy-master.zip	/.../caches/file_cache.py 73	
15	25	-	Python files within followy-master.zip	/followy-master/.../hash.py 54	
26	26	-	Python files within followy-master.zip	/followy-master/.../hashes.py 42	

Flaw Id	Module #	Class #	Module	Location	Fix By
61	27	-	Python files within followy-master.zip	/followy-master/.../Lib/hmac.py 52	
58	28	-	Python files within followy-master.zip	/followy-master/.../http.py 702	
8	29	-	Python files within followy-master.zip	/followy-master/.../index.py 277	
23	32	-	Python files within followy-master.zip	/.../util/langhelpers.py 29	
50	34	-	Python files within followy-master.zip	/followy-master/.../loaders.py 459	
56	35	-	Python files within followy-master.zip	/.../setuptools/package_index.py 264	
41	40	-	Python files within followy-master.zip	/.../werkzeug/contrib/sessions.py 83	
12	47	-	Python files within followy-master.zip	/followy-master/.../wheel/util.py 84	
16	51	-	Python files within followy-master.zip	/followy-master/.../pip/wheel.py 101	
60	51	-	Python files within followy-master.zip	/followy-master/.../pip/wheel.py 153	

Directory Traversal(14 flaws)

Description

Allowing user input to control paths used in filesystem operations may enable an attacker to access or modify otherwise protected system resources that would normally be inaccessible to end users. In some cases, the user-provided input may be passed directly to the filesystem operation, or it may be concatenated to one or more fixed strings to construct a fully-qualified path.

When an application improperly cleanses special character sequences in user-supplied filenames, a path traversal (or directory traversal) vulnerability may occur. For example, an attacker could specify a filename such as "../../etc/passwd", which resolves to a file outside of the intended directory that the attacker would not normally be authorized to view.

Recommendations

Assume all user-supplied input is malicious. Validate all user-supplied input to ensure that it conforms to the expected format, using centralized data validation routines when possible. When using black lists, be sure that the sanitizing routine performs a sufficient number of iterations to remove all instances of disallowed characters and ensure that the end result is not dangerous.

Associated Flaws by CWE ID:

External Control of File Name or Path (CWE ID 73)(14 flaws)

Description

This call contains a path manipulation flaw. The argument to the function is a filename constructed using untrusted input. If an attacker is allowed to specify all or part of the filename, it may be possible to gain unauthorized access to files on the server, including those outside the webroot, that would be normally be inaccessible to end users. The level of exposure depends on the effectiveness of input validation routines, if any.

Effort to Fix: 2 - Implementation error. Fix is approx. 6-50 lines of code. 1 day to fix.

Recommendations

Validate all untrusted input to ensure that it conforms to the expected format, using centralized data validation routines when possible. When using black lists, be sure that the sanitizing routine performs a sufficient number of iterations to remove all instances of disallowed characters.

Instances found via Static Scan

Flaw Id	Module #	Class #	Module	Location	Fix By
65	3	-	Python files within followy-master.zip	/followy-master/.../_compat.py 435	
75	3	-	Python files within followy-master.zip	/followy-master/.../_compat.py 459	
51	7	-	Python files within followy-master.zip	/.../roboto/awsqueryrequest.py 420	
30	7	-	Python files within followy-master.zip	/.../roboto/awsqueryrequest.py 421	
53	7	-	Python files within followy-master.zip	/.../roboto/awsqueryrequest.py 423	
37	31	-	Python files within followy-master.zip	/followy-master/.../iobject.py 95	
67	31	-	Python files within followy-master.zip	/followy-master/.../iobject.py 100	
44	43	-	Python files within followy-master.zip	/.../botenv/Lib/tempfile.py 551	
20	43	-	Python files within followy-master.zip	/.../botenv/Lib/tempfile.py 554	
7	43	-	Python files within followy-master.zip	/.../botenv/Lib/tempfile.py 613	
21	43	-	Python files within followy-master.zip	/.../botenv/Lib/tempfile.py 623	
35	45	-	Python files within followy-master.zip	/followy-master/.../Lib/token.py 98	
57	45	-	Python files within followy-master.zip	/followy-master/.../Lib/token.py 117	
69	45	-	Python files within followy-master.zip	/followy-master/.../Lib/token.py 134	

Server Configuration(3 flaws)

Description

The application's supporting infrastructure, including web servers and application servers, can impact the security of the deployed application. Failing to lock down a server, for example, can result in information leaks via error pages, stack traces, or unnecessary files left in a web-accessible directory. Even though these servers are not part of the application codebase, they create insecurities in the environment which contribute to overall risk.

Recommendations

Remove all extraneous files, including demonstration applications and sample code, from production systems. Configure production servers with the minimum set of services required for the application to function, and ensure that information leaks do not occur via server-generated error pages.

Audit any third party dependencies or services that are deployed by default to ensure that they do not compromise the security of the application being supported.

Associated Flaws by CWE ID:

→ ### Selection of Less-Secure Algorithm During Negotiation ('Algorithm Downgrade') (CWE ID 757)(3 flaws)

Description

A protocol or its implementation supports interaction between multiple actors and allows those actors to negotiate which algorithm should be used as a protection mechanism such as encryption or authentication, but it does not select the strongest algorithm that is available to both parties.

Effort to Fix: 1 - Trivial implementation error. Fix is up to 5 lines of code. One hour or less to fix.

Recommendations

Do not support SSLv2 or weak SSL/TLS ciphers (i.e. 56-bit key length or less, or other inherent weaknesses).

Instances found via Static Scan

Flaw Id	Module #	Class #	Module	Location	Fix By
5	41	-	Python files within followy-master.zip	/followy-master/.../util/ssl_.py 239	
43	42	-	Python files within followy-master.zip	/followy-master/.../util/ssl_.py 250	
14	48	-	Python files within followy-master.zip	/followy-master/.../util.py 1294	

Low (0 flaws)

No flaws of this type were found

Very Low (0 flaws)

No flaws of this type were found

Info (0 flaws)

No flaws of this type were found

About Veracode's Methodology

The Veracode platform uses static and dynamic analysis (for web applications) to inspect executables and identify security flaws in your applications. Using both static and dynamic analysis helps reduce false negatives and detect a broader range of security flaws. The static binary analysis engine models the binary executable into an intermediate representation, which is then verified for security flaws using a set of automated security scans. Dynamic analysis uses an automated penetration testing technique to detect security flaws at runtime. Once the automated process is complete, a security technician verifies the output to ensure the lowest false positive rates in the industry. The end result is an accurate list of security flaws for the classes of automated scans applied to the application.

Veracode Rating System Using Multiple Analysis Techniques

Higher assurance applications require more comprehensive analysis to accurately score their security quality. Because each analysis technique (automated static, automated dynamic, manual penetration testing or manual review) has differing false negative (FN) rates for different types of security flaws, any single analysis technique or even combination of techniques is bound to produce a certain level of false negatives. Some false negatives are acceptable for lower business critical applications, so a less expensive analysis using only one or two analysis techniques is acceptable. At higher business criticality the FN rate should be close to zero, so multiple analysis techniques are recommended.

Application Security Policies

The Veracode platform allows an organization to define and enforce a uniform application security policy across all applications in its portfolio. The elements of an application security policy include the target Veracode Level for the application; types of flaws that should not be in the application (which may be defined by flaw severity, flaw category, CWE, or a common standard including OWASP, CWE/SANS Top 25, or PCI); minimum Veracode security score; required scan types and frequencies; and grace period within which any policy-relevant flaws should be fixed.

Policy constraints

Policies have three main constraints that can be applied: rules, required scans, and remediation grace periods.

Evaluating applications against a policy

When an application is evaluated against a policy, it can receive one of four assessments:

Not assessed The application has not yet had a scan published
Passed The application has passed all the aspects of the policy, including rules, required scans, and grace period.
Did not pass The application has not completed all required scans; has not achieved the target Veracode Level; or has one or more policy relevant flaws that have exceeded the grace period to fix.
Conditional pass The application has one or more policy relevant flaws that have not yet exceeded the grace period to fix.

Understand Veracode Levels

The Veracode Level (VL) achieved by an application is determined by type of testing performed on the application, and the severity and types of flaws detected. A minimum security score (defined below) is also required for each level.

There are five Veracode Levels denoted as VL1, VL2, VL3, VL4, and VL5. VL1 is the lowest level and is achieved by demonstrating that security testing, automated static or dynamic, is utilized during the SDLC. VL5 is the highest level and is achieved by performing automated and manual testing and removing all significant flaws. The Veracode Levels VL2, VL3, and VL4 form a continuum of increasing software assurance between VL1 and VL5.

For IT staff operating applications, Veracode Levels can be used to set application security policies. For deployment scenarios of different business criticality, differing VLs should be made requirements. For example, the policy for applications that handle credit card transactions, and therefore have PCI compliance requirements, should be VL5. A medium business criticality internal application could have a policy requiring VL3.

Software developers can decide which VL they want to achieve based on the requirements of their customers. Developers of software that is mission critical to most of their customers will want to achieve VL5. Developers of general purpose business software may want to achieve VL3 or VL4. Once the software has achieved a Veracode Level it can be communicated to customers through a Veracode Report or through the Veracode Directory on the Veracode web site.

Criteria for achieving Veracode Levels

The following table defines the details to achieve each Veracode Level. The criteria for all columns: Flaw Severities Not Allowed, Flaw Categories not Allowed, Testing Required, and Minimum Score.

*Dynamic is only an option for web applications.

Veracode Level	Flaw Severities Not Allowed	Testing Required*	Minimum Score
VL5	V.High, High, Medium	Static AND Manual	90
VL4	V.High, High, Medium	Static	80
VL3	V.High, High	Static	70
VL2	V.High	Static OR Dynamic OR Manual	60
VL1		Static OR Dynamic OR Manual	

When multiple testing techniques are used it is likely that not all testing will be performed on the exact same build. If that is the case the latest test results from a particular technique will be used to calculate the current Veracode Level. After 6 months test results will be deemed out of date and will no longer be used to calculate the current Veracode Level.

Business Criticality

The foundation of the Veracode rating system is the concept that more critical applications require higher security quality scores to be acceptable risks. Less business critical applications can tolerate lower security quality. The business criticality is dictated by the typical deployed environment and the value of data used by the application. Factors that determine business criticality are: reputation damage, financial loss, operational risk, sensitive information disclosure, personal safety, and legal violations.

US. Govt. OMB Memorandum M-04-04; NIST FIPS Pub. 199

Business Criticality	Description
Very High	Mission critical for business/safety of life and limb on the line
High	Exploitation causes serious brand damage and financial loss with long term business impact
Medium	Applications connected to the internet that process financial or private customer information
Low	Typically internal applications with non-critical business impact
Very Low	Applications with no material business impact

Business Criticality Definitions

Very High (BC5) This is typically an application where the safety of life or limb is dependent on the system; it is mission critical the application maintain 100% availability for the long term viability of the project or business. Examples are control software for industrial, transportation or medical equipment or critical business systems such as financial trading systems.
High (BC4) This is typically an important multi-user business application reachable from the internet and is critical that the application maintain high availability to accomplish its mission. Exploitation of high criticality applications cause serious brand damage and business/financial loss and could lead to long term business impact.
Medium (BC3) This is typically a multi-user application connected to the internet or any system that processes financial or private customer information. Exploitation of medium criticality applications typically result in material business impact resulting

in some financial loss, brand damage or business liability. An example is a financial services company's internal 401K management system.
Low (BC2) This is typically an internal only application that requires low levels of application security such as authentication to protect access to non-critical business information and prevent IT disruptions. Exploitation of low criticality applications may lead to minor levels of inconvenience, distress or IT disruption. An example internal system is a conference room reservation or business card order system.
Very Low (BC1) Applications that have no material business impact should its confidentiality, data integrity and availability be affected. Code security analysis is not required for applications at this business criticality, and security spending should be directed to other higher criticality applications.

Scoring Methodology

The Veracode scoring system, Security Quality Score, is built on the foundation of two industry standards, the Common Weakness Enumeration (CWE) and Common Vulnerability Scoring System (CVSS). CWE provides the dictionary of security flaws and CVSS provides the foundation for computing severity, based on the potential Confidentiality, Integrity and Availability impact of a flaw if exploited.

The Security Quality Score is a single score from 0 to 100, where 0 is the most insecure application and 100 is an application with no detectable security flaws. The score calculation includes non-linear factors so that, for instance, a single Severity 5 flaw is weighted more heavily than five Severity 1 flaws, and so that each additional flaw at a given severity contributes progressively less to the score.

Veracode assigns a severity level to each flaw type based on three foundational application security requirements — Confidentiality, Integrity and Availability. Each of the severity levels reflects the potential business impact if a security breach occurs across one or more of these security dimensions.

Confidentiality Impact

According to CVSS, this metric measures the impact on confidentiality if a exploit should occur using the vulnerability on the target system. At the weakness level, the scope of the Confidentiality in this model is within an application and is measured at three levels of impact -None, Partial and Complete.

Integrity Impact

This metric measures the potential impact on integrity of the application being analyzed. Integrity refers to the trustworthiness and guaranteed veracity of information within the application. Integrity measures are meant to protect data from unauthorized modification. When the integrity of a system is sound, it is fully proof from unauthorized modification of its contents.

Availability Impact

This metric measures the potential impact on availability if a successful exploit of the vulnerability is carried out on a target application. Availability refers to the accessibility of information resources. Almost exclusive to this domain are denial-of-service vulnerabilities. Attacks that compromise authentication and authorization for application access, application memory, and administrative privileges are examples of impact on the availability of an application.

Security Quality Score Calculation

The overall Security Quality Score is computed by aggregating impact levels of all weaknesses within an application and representing the score on a 100 point scale. This score does not predict vulnerability potential as much as it enumerates the security weaknesses and their impact levels within the application code.

The Raw Score formula puts weights on each flaw based on its impact level. These weights are exponential and determined by empirical analysis by Veracode's application security experts with validation from industry experts. The score is normalized to a scale of 0 to 100, where a score of 100 is an application with 0 detected flaws using the analysis technique for the application's business criticality.

Understand Severity, Exploitability, and Remediation Effort

Severity and exploitability are two different measures of the seriousness of a flaw. Severity is defined in terms of the potential impact to confidentiality, integrity, and availability of the application as defined in the CVSS, and exploitability is defined in terms of the likelihood or ease with which a flaw can be exploited. A high severity flaw with a high likelihood of being exploited by an attacker is potentially more dangerous than a high severity flaw with a low likelihood of being exploited.

Remediation effort, also called Complexity of Fix, is a measure of the likely effort required to fix a flaw. Together with severity, the remediation effort is used to give Fix First guidance to the developer.

Veracode Flaw Severities

Veracode flaw severities are defined as follows:

Severity	Description
Very High	The offending line or lines of code is a very serious weakness and is an easy target for an attacker. The code should be modified immediately to avoid potential attacks.
High	The offending line or lines of code have significant weakness, and the code should be modified immediately to avoid potential attacks.
Medium	A weakness of average severity. These should be fixed in high assurance software. A fix for this weakness should be considered after fixing the very high and high for medium assurance software.
Low	This is a low priority weakness that will have a small impact on the security of the software. Fixing should be consideration for high assurance software. Medium and low assurance software can ignore these flaws.
Very Low	Minor problems that some high assurance software may want to be aware of. These flaws can be safely ignored in medium and low assurance software.
Informational	Issues that have no impact on the security quality of the application but which may be of interest to the reviewer.

Informational findings

Informational severity findings are items observed in the analysis of the application that have no impact on the security quality of the application but may be interesting to the reviewer for other reasons. These findings may include code quality issues, API usage, and other factors.

Informational severity findings have no impact on the security quality score of the application and are not included in the summary tables of flaws for the application.

Exploitability

Each flaw instance in a static scan may receive an exploitability rating. The rating is an indication of the intrinsic likelihood that the flaw may be exploited by an attacker. Veracode recommends that the exploitability rating be used to prioritize flaw remediation within a particular group of flaws with the same severity and difficulty of fix classification.

The possible exploitability ratings include:

Exploitability	Description
V. Unlikely	Very unlikely to be exploited
Unlikely	Unlikely to be exploited

Exploitability	Description
Neutral	Neither likely nor unlikely to be exploited.
Likely	Likely to be exploited
V. Likely	Very likely to be exploited

Note: All reported flaws found via dynamic scans are assumed to be exploitable, because the dynamic scan actually executes the attack in question and verifies that it is valid.

Effort/Complexity of Fix

Each flaw instance receives an effort/complexity of fix rating based on the classification of the flaw. The effort/complexity of fix rating is given on a scale of 1 to 5, as follows:

Effort/Complexity of Fix	Description
5	Complex design error. Requires significant redesign.
4	Simple design error. Requires redesign and up to 5 days to fix.
3	Complex implementation error. Fix is approx. 51-500 lines of code. Up to 5 days to fix.
2	Implementation error. Fix is approx. 6-50 lines of code. 1 day to fix.
1	Trivial implementation error. Fix is up to 5 lines of code. One hour or less to fix.

Flaw Types by Severity Level

The flaw types by severity level table provides a summary of flaws found in the application by Severity and Category. The table puts the Security Quality Score into context by showing the specific breakout of flaws by severity, used to compute the score as described above. If multiple analysis techniques are used, the table includes a breakout of all flaws by category and severity for each analysis type performed.

Flaws by Severity

The flaws by severity chart shows the distribution of flaws by severity. An application can get a mediocre security rating by having a few high risk flaws or many medium risk flaws.

Flaws in Common Modules

The flaws in common modules listing shows a summary of flaws in shared dependency modules in this application. A shared dependency is a dependency that is used by more than one analyzed module. Each module is listed with the number of executables that consume it as a dependency and a summary of the impact on the application's security score of the flaws found in the dependency.

The score impact represents the amount that the application score would increase if all the flaws in the shared dependency module were fixed. This information can be used to focus remediation efforts on common modules with a higher impact on the application security score.

Only common modules that were uploaded with debug information are included in the Flaws in Common Modules listing.

Action Items

The Action Items section of the report provides guidance on the steps required to bring the application to a state where it passes its assigned policy. These steps may include fixing or mitigating flaws or performing additional scans. The section also includes best practice recommendations to improve the security quality of the application.

Common Weakness Enumeration (CWE)

The Common Weakness Enumeration (CWE) is an industry standard classification of types of software weaknesses, or flaws, that can lead to security problems. CWE is widely used to provide a standard taxonomy of software errors. Every flaw in a Veracode report is classified according to a standard CWE identifier.

More guidance and background about the CWE is available at http://cwe.mitre.org/data/index.html.

About Manual Assessments

The Veracode platform can include the results from a manual assessment (usually a penetration test or code review) as part of a report. These results differ from the results of automated scans in several important ways, including objectives, attack vectors, and common attack patterns.

A manual penetration assessment is conducted to observe the application code in a run-time environment and to simulate real-world attack scenarios. Manual testing is able to identify design flaws, evaluate environmental conditions, compound multiple lower risk flaws into higher risk vulnerabilities, and determine if identified flaws affect the confidentiality, integrity, or availability of the application.

Objectives

The stated objectives of a manual penetration assessment are:

- Perform testing, using proprietary and/or public tools, to determine whether it is possible for an attacker to:
- Circumvent authentication and authorization mechanisms
- Escalate application user privileges
- Hijack accounts belonging to other users
- Violate access controls placed by the site administrator
- Alter data or data presentation
- Corrupt application and data integrity, functionality and performance
- Circumvent application business logic
- Circumvent application session management
- Break or analyze use of cryptography within user accessible components
- Determine possible extent access or impact to the system by attempting to exploit vulnerabilities
- Score vulnerabilities using the Common Vulnerability Scoring System (CVSS)
- Provide tactical recommendations to address security issues of immediate consequence

Provide strategic recommendations to enhance security by leveraging industry best practices

Attack vectors

In order to achieve the stated objectives, the following tests are performed as part of the manual penetration assessment, when applicable to the platforms and technologies in use:

- Cross Site Scripting (XSS)
- SQL Injection
- Command Injection
- Cross Site Request Forgery (CSRF)
- Authentication/Authorization Bypass
- Session Management testing, e.g. token analysis, session expiration, and logout effectiveness
- Account Management testing, e.g. password strength, password reset, account lockout, etc.
- Directory Traversal
- Response Splitting
- Stack/Heap Overflows
- Format String Attacks

- Cookie Analysis
- Server Side Includes Injection
- Remote File Inclusion
- LDAP Injection
- XPATH Injection
- Internationalization attacks
- Denial of Service testing at the application layer only
- AJAX Endpoint Analysis
- Web Services Endpoint Analysis
- HTTP Method Analysis
- SSL Certificate and Cipher Strength Analysis
- Forced Browsing

CAPEC Attack Pattern Classification

The following attack pattern classifications are used to group similar application flaws discovered during manual penetration testing. Attack patterns describe the general methods employed to access and exploit the specific weaknesses that exist within an application. CAPEC (Common Attack Pattern Enumeration and Classification) is an effort led by Cigital, Inc. and is sponsored by the United States Department of Homeland Security's National Cyber Security Division.

Abuse of Functionality

Exploitation of business logic errors or misappropriation of programmatic resources. Application functions are developed to specifications with particular intentions, and these types of attacks serve to undermine those intentions.

Examples:

- Exploiting password recovery mechanisms
- Accessing unpublished or test APIs
- Cache poisoning

Spoofing

Impersonation of entities or trusted resources. A successful attack will present itself to a verifying entity with an acceptable level of authenticity.

Examples:

- Man in the middle attacks
- Checksum spoofing
- Phishing attacks

Probabilistic Techniques

Using predictive capabilities or exhaustive search techniques in order to derive or manipulate sensitive information. Attacks capitalize on the availability of computing resources or the lack of entropy within targeted components.

Examples:

- Password brute forcing
- Cryptanalysis
- Manipulation of authentication tokens

Exploitation of Authentication

Circumventing authentication requirements to access protected resources. Design or implementation flaws may allow authentication checks to be ignored, delegated, or bypassed.

Examples:

- Cross-site request forgery
- Reuse of session identifiers
- Flawed authentication protocol

Resource Depletion

Affecting the availability of application components or resources through symmetric or asymmetric consumption. Unrestricted access to computationally expensive functions or implementation flaws that affect the stability of the application can be targeted by an attacker in order to cause denial of service conditions.

Examples:

- Flooding attacks
- Unlimited file upload size
- Memory leaks

Exploitation of Privilege/Trust

Undermining the application's trust model in order to gain access to protected resources or gain additional levels of access as defined by the application. Applications that implicitly extend trust to resources or entities outside of their direct control are susceptible to attack.

Examples:

- Insufficient access control lists
- Circumvention of client side protections
- Manipulation of role identification information

Injection

Inserting unexpected inputs to manipulate control flow or alter normal business processing. Applications must contain sufficient data validation checks in order to sanitize tainted data and prevent malicious, external control over internal processing.

Examples:

- SQL Injection
- Cross-site scripting
- XML Injection

Data Structure Attacks

Supplying unexpected or excessive data that results in more data being written to a buffer than it is capable of holding. Successful attacks of this class can result in arbitrary command execution or denial of service conditions.

Examples:

- Buffer overflow
- Integer overflow
- Format string overflow

Data Leakage Attacks

Recovering information exposed by the application that may itself be confidential or may be useful to an attacker in discovering or exploiting other weaknesses. A successful attack may be conducted passive observation or active interception methods. This attack pattern often manifests itself in the form of applications that expose sensitive information within error messages.

Examples:

- Sniffing clear-text communication protocols
- Stack traces returned to end users
- Sensitive information in HTML comments

Resource Manipulation

Manipulating application dependencies or accessed resources in order to undermine security controls and gain unauthorized access to protected resources. Applications may use tainted data when constructing paths to local resources or when constructing processing environments.

Examples:

- Carriage Return Line Feed log file injection
- File retrieval via path manipulation
- User specification of configuration files

Time and State Attacks

Undermining state condition assumptions made by the application or capitalizing on time delays between security checks and performed operations. An application that does not enforce a required processing sequence or does not handle concurrency adequately will be susceptible to these attack patterns.

Examples:

- Bypassing intermediate form processing steps
- Time-of-check and time-of-use race conditions
- Deadlock triggering to cause a denial of service

Terms of Use

Use and distribution of this report are governed by the agreement between Veracode and its customer. In particular, this report and the results in the report cannot be used publicly in connection with Veracode's name without written permission.

Appendix A: Referenced Source Files

Id	Filename	Path
1	__init__.py	/followy-master/botenv/Lib/site-packages/werkzeug/debug/
2	__init__.py	/followy-master/botenv/Lib/site-packages/oauthlib/oauth1/rfc5849/
3	_compat.py	/followy-master/botenv/Lib/site-packages/click/
4	arbiter.py	/followy-master/botenv/Lib/site-packages/gunicorn/
5	auth.py	/followy-master/botenv/Lib/site-packages/pip/_vendor/requests/
6	auth.py	/followy-master/botenv/Lib/site-packages/requests/
7	awsqueryrequest.py	/followy-master/botenv/Lib/site-packages/boto/roboto/
8	base.py	/followy-master/botenv/Lib/site-packages/gunicorn/workers/
9	bccache.py	/followy-master/botenv/Lib/site-packages/jinja2/
10	bot.py	/followy-master/
11	cache.py	/followy-master/botenv/Lib/site-packages/tweepy/
12	cache.py	/followy-master/botenv/Lib/site-packages/werkzeug/contrib/
13	connection.py	/followy-master/botenv/Lib/site-packages/boto/sns/
14	connection.py	/followy-master/botenv/Lib/site-packages/boto/route53/
15	connection.py	/followy-master/botenv/Lib/site-packages/boto/
16	connection.py	/followy-master/botenv/Lib/site-packages/boto/mws/
17	cx_oracle.py	/followy-master/botenv/Lib/site-packages/sqlalchemy/dialects/oracle/
18	database.py	/followy-master/botenv/Lib/site-packages/pip/_vendor/distlib/
19	debugger.js	/followy-master/botenv/Lib/site-packages/werkzeug/debug/shared/
20	default.py	/followy-master/botenv/Lib/site-packages/sqlalchemy/engine/
21	djbec.py	/followy-master/botenv/Lib/site-packages/wheel/signatures/
22	easy_install.py	/followy-master/botenv/Lib/site-packages/setuptools/command/
23	file_cache.py	/followy-master/botenv/Lib/site-packages/pip/_vendor/cachecontrol/caches/
24	filters.py	/followy-master/botenv/Lib/site-packages/jinja2/
25	hash.py	/followy-master/botenv/Lib/site-packages/pip/commands/
26	hashes.py	/followy-master/botenv/Lib/site-packages/pip/utils/
27	hmac.py	/followy-master/botenv/Lib/
28	http.py	/followy-master/botenv/Lib/site-packages/werkzeug/
29	index.py	/followy-master/botenv/Lib/site-packages/pip/_vendor/distlib/
30	installer.py	/followy-master/botenv/Lib/site-packages/boto/pyami/installers/ubuntu/
31	iobject.py	/followy-master/botenv/Lib/site-packages/boto/mashups/
32	langhelpers.py	/followy-master/botenv/Lib/site-packages/sqlalchemy/util/
33	launch.py	/followy-master/botenv/Lib/site-packages/setuptools/
34	loaders.py	/followy-master/botenv/Lib/site-packages/jinja2/
35	package_index.py	/followy-master/botenv/Lib/site-packages/setuptools/
36	posixemulation.py	/followy-master/botenv/Lib/site-packages/werkzeug/
37	resumable_upload_handler.py	/followy-master/botenv/Lib/site-packages/boto/gs/
38	retrying.py	/followy-master/botenv/Lib/site-packages/pip/_vendor/

Id	Filename	Path
39	serving.py	/followy-master/botenv/Lib/site-packages/werkzeug/
40	sessions.py	/followy-master/botenv/Lib/site-packages/werkzeug/contrib/
41	ssl_.py	/followy-master/botenv/Lib/site-packages/pip/_vendor/requests/packages/urllib3/util/
42	ssl_.py	/followy-master/botenv/Lib/site-packages/urllib3/util/
43	tempfile.py	/followy-master/botenv/Lib/
44	test.py	/followy-master/botenv/Lib/site-packages/werkzeug/
45	token.py	/followy-master/botenv/Lib/
46	util.py	/followy-master/botenv/Lib/site-packages/sqlalchemy/testing/
47	util.py	/followy-master/botenv/Lib/site-packages/wheel/
48	util.py	/followy-master/botenv/Lib/site-packages/pip/_vendor/distlib/
49	utils.py	/followy-master/botenv/Lib/site-packages/jinja2/
50	utils.py	/followy-master/botenv/Lib/site-packages/boto/
51	wheel.py	/followy-master/botenv/Lib/site-packages/pip/

Appendix B: Dynamic Flaw Inventory

Rescan Status	Number of Flaws
All	0
New	0
Open and Reopened	0
Cannot Reproduce	0
Fixed	0