Fortify Audit Workbench

Developer Workbook

boomerang4_0

Table of Contents

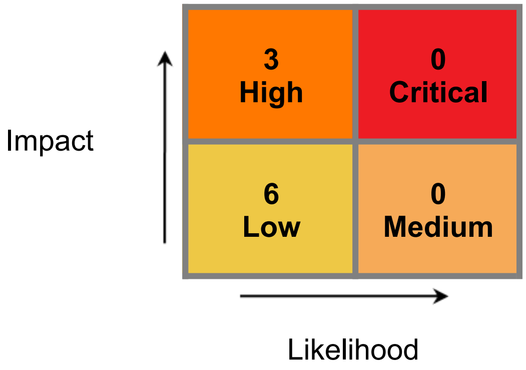
Executive Summary

This workbook is intended to provide all necessary details and information for a developer to understand and remediate the different issues discovered during the boomerang4_0 project audit. The information contained in this workbook is targeted at project managers and developers.

This section provides an overview of the issues uncovered during analysis.

Project Name:	boomerang4_0
Project Version:	
SCA:	Results Present
WebInspect:	Results Not Present
WebInspect Agent:	Results Not Present
Other:	Results Not Present

Issues by Priority



3 High	0 Critical
6 Low	0 Medium

Top Ten Critical Categories

This project does not contain any critical issues

Project Description

This section provides an overview of the Fortify scan engines used for this project, as well as the project meta-information.

SCA

Date of Last Analysis:	Sep 1, 2020, 11:44 AM	**Engine Version:**	20.1.0.0158
Host Name:	wbgmsois001	**Certification:**	VALID
Number of Files:	10	**Lines of Code:**	2,167

Rulepack Name	Rulepack Version
Fortify Secure Coding Rules, Core, JavaScript	2020.2.0.0010
Fortify Secure Coding Rules, Extended, Configuration	2020.2.0.0010
Fortify Secure Coding Rules, Extended, Content	2020.2.0.0010
Fortify Secure Coding Rules, Extended, JavaScript	2020.2.0.0010

Issue Breakdown by Fortify Categories

The following table depicts a summary of all issues grouped vertically by Fortify Category. For each category, the total number of issues is shown by Fortify Priority Order, including information about the number of audited issues.

Category	Fortify Priority (audited/total)				Total Issues
	Critical	High	Medium	Low	
Cross-Site Request Forgery	0	0	0	0 / 1	0 / 1
Insecure Randomness	0	0 / 3	0	0	0 / 3
Weak Cryptographic Hash	0	0	0	0 / 5	0 / 5

Results Outline

Cross-Site Request Forgery (1 issue)

Abstract

HTTP requests must contain a user-specific secret in order to prevent an attacker from making unauthorized requests.

Explanation

A cross-site request forgery (CSRF) vulnerability occurs when: 1. A web application uses session cookies. 2. The application acts on an HTTP request without verifying that the request was made with the user's consent. A nonce is a cryptographic random value that is sent with a message to prevent replay attacks. If the request does not contain a nonce that proves its provenance, the code that handles the request is vulnerable to a CSRF attack (unless it does not change the state of the application). This means a web application that uses session cookies has to take special precautions in order to ensure that an attacker can't trick users into submitting bogus requests. Imagine a web application that allows administrators to create new accounts as follows:

```
  var req = new XMLHttpRequest();
  req.open("POST", "/new_user", true);
  body = addToPost(body, new_username);
  body = addToPost(body, new_passwd);
  req.send(body);
```

An attacker might set up a malicious web site that contains the following code.

```
  var req = new XMLHttpRequest();
  req.open("POST", "http://www.example.com/new_user", true);
  body = addToPost(body, "attacker");
  body = addToPost(body, "haha");
  req.send(body);
```

If an administrator for `example.com` visits the malicious page while she has an active session on the site, she will unwittingly create an account for the attacker. This is a CSRF attack. It is possible because the application does not have a way to determine the provenance of the request. Any request could be a legitimate action chosen by the user or a faked action set up by an attacker. The attacker does not get to see the Web page that the bogus request generates, so the attack technique is only useful for requests that alter the state of the application. Applications that pass the session identifier in the URL rather than as a cookie do not have CSRF problems because there is no way for the attacker to access the session identifier and include it as part of the bogus request. CSRF is entry number five on the 2007 OWASP Top 10 list.

Recommendation

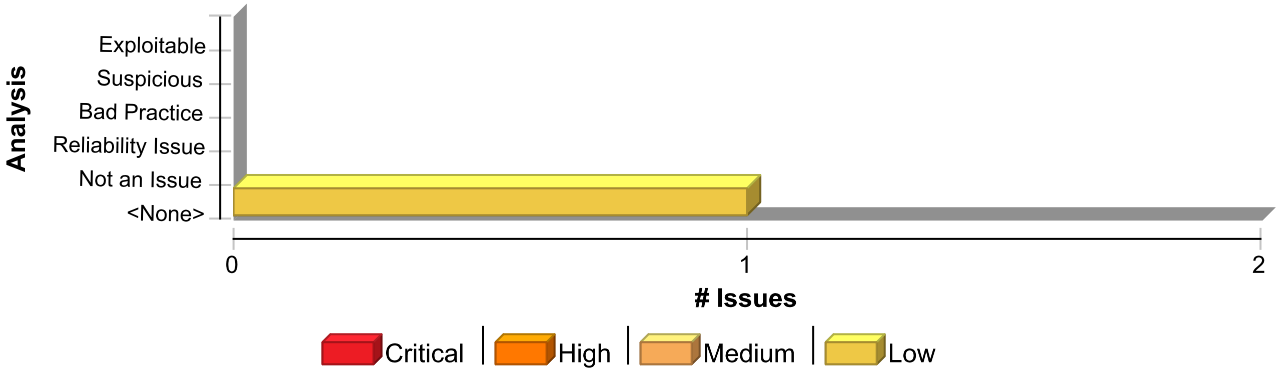
Applications that use session cookies must include some piece of information in every form post that the back-end code can use to validate the provenance of the request. One way to do that is to include a random request identifier or nonce, as follows:

```
  RequestBuilder rb = new RequestBuilder(RequestBuilder.POST, "/new_user");
  body = addToPost(body, new_username);
  body = addToPost(body, new_passwd);
  body = addToPost(body, request_id);
  rb.sendRequest(body, new NewAccountCallback(callback));
```

Then the back-end logic can validate the request identifier before processing the rest of the form data. When possible, the request identifier should be unique to each server request rather than shared across every request for a particular session. As with session identifiers, the harder it is for an attacker to guess the request identifier, the harder it is to conduct a successful CSRF attack. The token should not be easily guessed and it should be protected in the same way that session tokens are protected, such as using

SSLv3. Additional mitigation techniques include: **Framework protection:** Most modern web application frameworks embed CSRF protection and they will automatically include and verify CSRF tokens. **Use a Challenge-Response control:** Forcing the customer to respond to a challenge sent by the server is a strong defense against CSRF. Some of the challenges that can be used for this purpose are: CAPTCHAs, password re-authentication and one-time tokens. **Check HTTP Referer/Origin headers:** An attacker won't be able to spoof these headers while performing a CSRF attack. This makes these headers a useful method to prevent CSRF attacks. **Double-submit Session Cookie:** Sending the session ID Cookie as a hidden form value in addition to the actual session ID Cookie is a good protection against CSRF attacks. The server will check both values and make sure they are identical before processing the rest of the form data. If an attacker submits a form in behalf of a user, he won't be able to modify the session ID cookie value as per the same-origin-policy. **Limit Session Lifetime:** When accessing protected resources using a CSRF attack, the attack will only be valid as long as the session ID sent as part of the attack is still valid on the server. Limiting the Session lifetime will reduce the probability of a successful attack. The techniques described here can be defeated with XSS attacks. Effective CSRF mitigation includes XSS mitigation techniques.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Cross-Site Request Forgery	1	0	0	1
Total	**1**	**0**	**0**	**1**

Cross-Site Request Forgery — Low

Package: js

js/boomerang.js, line 3542 (Cross-Site Request Forgery) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionPointerCall: open
Enclosing Method: sendXhrPostBeacon()
File: js/boomerang.js:3542
Taint Flags:

```
3539  * @memberof BOOMR
3540  */
```

Cross-Site Request Forgery — Low

Package: js

js/boomerang.js, line 3542 (Cross-Site Request Forgery) — Low

```
3541  sendXhrPostBeacon: function(xhr, paramsJoined) {
3542  xhr.open("POST", impl.beacon_url);
3543
3544  xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
3545
```

Insecure Randomness (3 issues)

Abstract

Standard pseudorandom number generators cannot withstand cryptographic attacks.

Explanation

Insecure randomness errors occur when a function that can produce predictable values is used as a source of randomness in a security-sensitive context. Computers are deterministic machines, and as such are unable to produce true randomness. Pseudorandom Number Generators (PRNGs) approximate randomness algorithmically, starting with a seed from which subsequent values are calculated. There are two types of PRNGs: statistical and cryptographic. Statistical PRNGs provide useful statistical properties, but their output is highly predictable and form an easy to reproduce numeric stream that is unsuitable for use in cases where security depends on generated values being unpredictable. Cryptographic PRNGs address this problem by generating output that is more difficult to predict. For a value to be cryptographically secure, it must be impossible or highly improbable for an attacker to distinguish between the generated random value and a truly random value. In general, if a PRNG algorithm is not advertised as being cryptographically secure, then it is probably a statistical PRNG and should not be used in security-sensitive contexts, where its use can lead to serious vulnerabilities such as easy-to-guess temporary passwords, predictable cryptographic keys, session hijacking, and DNS spoofing. **Example:** The following code uses a statistical PRNG to create a URL for a receipt that remains active for some period of time after a purchase.

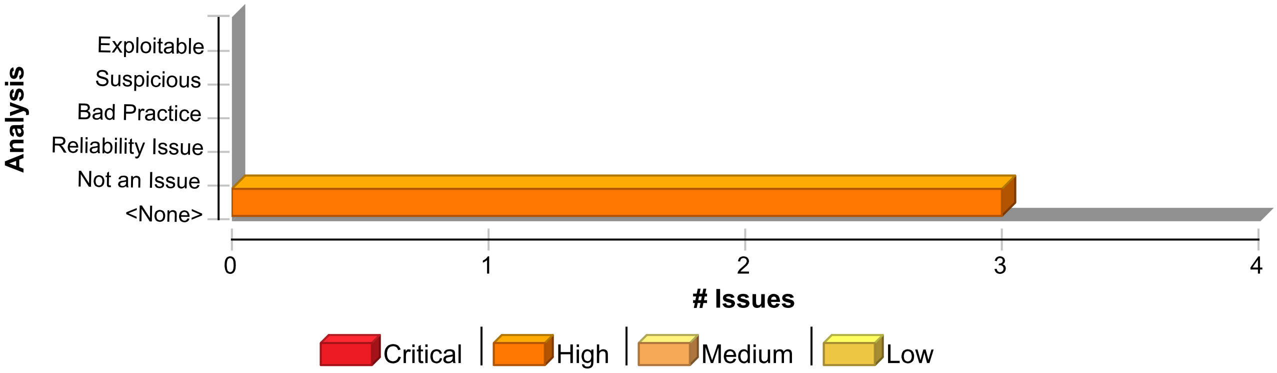
```
function genReceiptURL (baseURL){
  var randNum = Math.random();
  var receiptURL = baseURL + randNum + ".html";
  return receiptURL;
}
```

This code uses the `Math.random()` function to generate "unique" identifiers for the receipt pages it generates. Since `Math.random()` is a statistical PRNG, it is easy for an attacker to guess the strings it generates. Although the underlying design of the receipt system is also faulty, it would be more secure if it used a random number generator that did not produce predictable receipt identifiers, such as a cryptographic PRNG.

Recommendation

When unpredictability is critical, as is the case with most security-sensitive uses of randomness, use a cryptographic PRNG. Regardless of the PRNG you choose, always use a value with sufficient entropy to seed the algorithm. (Do not use values such as the current time because it offers only negligible entropy.) In JavaScript, the typical recommendation is to use the `window.crypto.random()` function in the Mozilla API. However, this method does not work in many browsers, including more recent versions of Mozilla Firefox. There is currently no cross-browser solution for a robust cryptographic PRNG. In the meantime, consider handling any PRNG functionality outside of JavaScript.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Insecure Randomness	3	0	0	3
Total	**3**	**0**	**0**	**3**

Insecure Randomness — High

Package: js

js/boomerang.js, line 1023 (Insecure Randomness) — High

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionPointerCall: random
Enclosing Method: lambda()
File: js/boomerang.js:1023
Taint Flags:

```
1020  *
1021  * @memberof BOOMR.session
1022  */
1023  ID: Math.random().toString(36).replace(/^0\./, ""),
1024
1025  /**
1026  * Session start time.
```

js/boomerang.js, line 1824 (Insecure Randomness) — High

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionPointerCall: random
Enclosing Method: lambda()
File: js/boomerang.js:1824
Taint Flags:

Insecure Randomness — High

Package: js

js/boomerang.js, line 1824 (Insecure Randomness) — High

```
1821  */
1822  generateUUID: function() {
1823  return "xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx".replace(/[xy]/g, function(c) {
1824  var r = Math.random() * 16 | 0;
1825  var v = c === "x" ? r : (r & 0x3 | 0x8);
1826  return v.toString(16);
1827  });
```

js/boomerang.js, line 1842 (Insecure Randomness) — High

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionPointerCall: random
Enclosing Method: lambda()
File: js/boomerang.js:1842
Taint Flags:

```
1839  */
1840  generateId: function(chars) {
1841  return "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx".substr(0, chars || 40).replace(/x/g,
function(c) {
1842  var c = (Math.random() || 0.01).toString(36);
1843
1844  // some implementations may return "0" for small numbers
1845  if (c === "0") {
```

Weak Cryptographic Hash (5 issues)

Abstract

Weak cryptographic hashes cannot guarantee data integrity and should not be used in security-critical contexts.

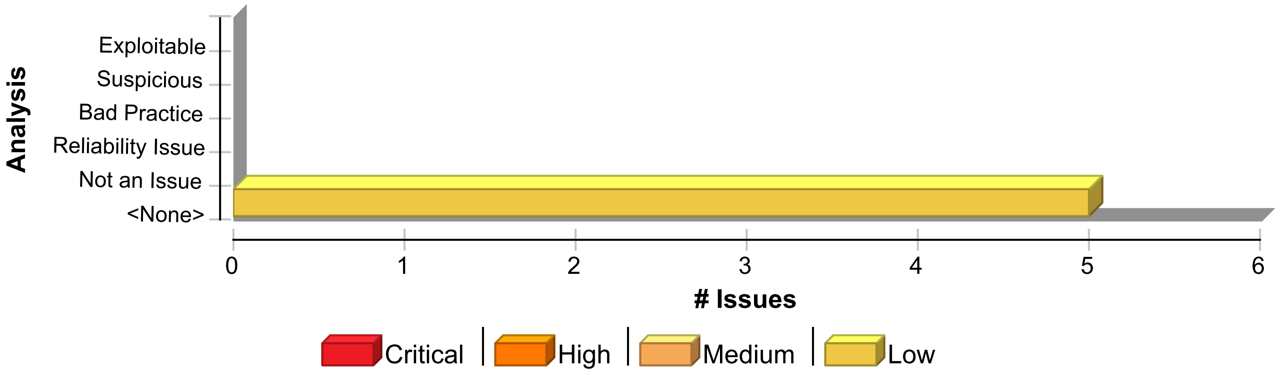
Explanation

MD2, MD4, MD5, RIPEMD-160, and SHA-1 are popular cryptographic hash algorithms often used to verify the integrity of messages and other data. However, as recent cryptanalysis research has revealed fundamental weaknesses in these algorithms, they should no longer be used within security-critical contexts. Effective techniques for breaking MD and RIPEMD hashes are widely available, so those algorithms should not be relied upon for security. In the case of SHA-1, current techniques still require a significant amount of computational power and are more difficult to implement. However, attackers have found the Achilles' heel for the algorithm, and techniques for breaking it will likely lead to the discovery of even faster attacks.

Recommendation

Discontinue the use of MD2, MD4, MD5, RIPEMD-160, and SHA-1 for data-verification in security-critical contexts. Currently, SHA-224, SHA-256, SHA-384, SHA-512, and SHA-3 are good alternatives. However, these variants of the Secure Hash Algorithm have not been scrutinized as closely as SHA-1, so be mindful of future research that might impact the security of these algorithms.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Weak Cryptographic Hash	5	0	0	5
Total	**5**	**0**	**0**	**5**

Weak Cryptographic Hash **Low**

Package: js

js/boomerang.js, line 1461 (Weak Cryptographic Hash) **Low**

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Weak Cryptographic Hash Low

Package: js

js/boomerang.js, line 1461 (Weak Cryptographic Hash) Low

Sink Details

Sink: FunctionPointerCall
Enclosing Method: lambda()
File: js/boomerang.js:1461
Taint Flags:

```
1458  return url;
1459  }
1460  return url.replace(/\?([^#]*)/, function(m0, m1) {
1461  return "?" + (m1.length > 10 ? BOOMR.utils.MD5(m1) : m1);
1462  });
1463  },
1464
```

js/wbgrt.js, line 1073 (Weak Cryptographic Hash) Low

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionPointerCall: MD5
Enclosing Method: _iterable_click()
File: js/wbgrt.js:1073
Taint Flags:

```
1070
1071  this.updateCookie(
1072  {
1073  "nu": BOOMR.utils.MD5(value)
1074  },
1075  "cl");
1076
```

js/wbgrt.js, line 1040 (Weak Cryptographic Hash) Low

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionPointerCall: MD5
Enclosing Method: page_unload()
File: js/wbgrt.js:1040
Taint Flags:

```
1037  //
1038  this.updateCookie(
```

Weak Cryptographic Hash

Low

Package: js

js/wbgrt.js, line 1040 (Weak Cryptographic Hash)

Low

```
1039  (!impl.navigationStart && impl.strict_referrer) ? {
1040  "r": BOOMR.utils.MD5(d.URL)
1041  } : null,
1042  edata.type === "beforeunload" ? "ul" : "hd"
1043  );
```

js/wbgrt.js, line 504 (Weak Cryptographic Hash)

Low

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionPointerCall: MD5
Enclosing Method: initFromCookie()
File: js/wbgrt.js:504
Taint Flags:

```
501  if (subcookies.s && (subcookies.r || subcookies.nu)) {
502  this.r = subcookies.r;
503  urlHash = BOOMR.utils.MD5(d.URL);
504  docReferrerHash = BOOMR.utils.MD5((d && d.referrer) || "");
505
506  // Either the URL of the page setting the cookie needs to match document.referrer
507  BOOMR.debug("referrer check: " + this.r + " =?= " + docReferrerHash, "rt");
```

js/wbgrt.js, line 503 (Weak Cryptographic Hash)

Low

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionPointerCall: MD5
Enclosing Method: initFromCookie()
File: js/wbgrt.js:503
Taint Flags:

```
500  // we check if the start time is usable.
501  if (subcookies.s && (subcookies.r || subcookies.nu)) {
502  this.r = subcookies.r;
503  urlHash = BOOMR.utils.MD5(d.URL);
504  docReferrerHash = BOOMR.utils.MD5((d && d.referrer) || "");
505
506  // Either the URL of the page setting the cookie needs to match document.referrer
```

Description of Key Terminology

Likelihood and Impact

Likelihood
Likelihood is the probability that a vulnerability will be accurately identified and successfully exploited.

Impact
Impact is the potential damage an attacker could do to assets by successfully exploiting a vulnerability. This damage can be in the form of, but not limited to, financial loss, compliance violation, loss of brand reputation, and negative publicity.

Fortify Priority Order

Critical
Critical-priority issues have high impact and high likelihood. Critical-priority issues are easy to detect and exploit and result in large asset damage. These issues represent the highest security risk to the application. As such, they should be remediated immediately.

SQL Injection is an example of a critical issue.

High
High-priority issues have high impact and low likelihood. High-priority issues are often difficult to detect and exploit, but can result in large asset damage. These issues represent a high security risk to the application. High-priority issues should be remediated in the next scheduled patch release.

Password Management: Hardcoded Password is an example of a high issue.

Medium
Medium-priority issues have low impact and high likelihood. Medium-priority issues are easy to detect and exploit, but typically result in small asset damage. These issues represent a moderate security risk to the application. Medium-priority issues should be remediated in the next scheduled product update.

Path Manipulation is an example of a medium issue.

Low
Low-priority issues have low impact and low likelihood. Low-priority issues can be difficult to detect and exploit and typically result in small asset damage. These issues represent a minor security risk to the application. Low-priority issues should be remediated as time allows.

Dead Code is an example of a low issue.