Paragon Initiative Enterprises

console.bytejail.com

Source Code Audit
2-3 May 2015

I. Introduction

This document describes the security audit of the Bytejail console server-side code by Paragon Initiative Enterprises for Christian Hermann (bitBeans).

Our audit targeted git commit `1c8b1c68432a9457edbd79b1a6d3bb68038bd4d0`, which was committed on May 1, 2015.

This report was prepared by Scott Arciszewski, CDO, and reviewed by Robyn Terjesen, CEO.

Audit Results Summary

After a comprehensive code review of the Bytejail console source code, we have identified three security issues: one of medium severity and two of low severity. All three are related to cryptography and believed to be difficult for an attacker to exploit.

Version History

1.0	2015-05-03	Initial report
1.1	2015-05-05	Verified that the issues we reported were corrected

II. Audit Scope

We have limited the scope of this audit to focus specifically on the Bytejail server-side application code and webserver software configuration.

We emphasized the following areas in our review:

- User registration
- User authentication
- Account recovery
- Two-Factor Authentication

We excluded the following from our scope:

- Client-side application code
- The Laravel framework
- The PHP interpreter, except where necessary
- The operating system
- Third-party libraries (e.g. openssl, libsodium) that were not developed for Bytejail

III. Issues

1. Weak Entropy in Password Reset

Severity: **Medium**
Exploitability: **Difficult**

Affects:

- `app/Helpers/Helpers.php`

When a password reset request is initiated, Bytejail will send a randomly generated password reset link to the user's inbox.

After clicking on the link, it will then send a new password to the same email address.

The code used to generate new passwords is:

```
function generate_password() {
    $chars = env('PASSWORD_CHARS');
    $password = substr( str_shuffle( $chars ), 0, env('PASSWORD_LENGTH') );
    return $password;
}
```

The environment variable PASSWORD_CHARS is set to a 64-character string with all unique characters and PASSWORD_LENGTH is 14.

Theoretically, this password construct would result in about 81 bits of entropy.

$_{64}P_{14} = 64!/49!$
$\log_2(64!/49!)$ is about 81.21

However, the actual entropy is much lower. PHP's `str_shuffle()` function internally shuffles the string by iteratively swapping out a constant character (starting from the last, working its way to index 0) with an index determined by the `rand()` function. Due to limitations of `rand()`, the entropy is therefore limited to, at most, 32 bits. (Ref. 1)

This means about a 2^{50} reduction to the brute force threshold for a freshly reset password.

To exploit this weakness, an attacker would need to first predict the internal state of `rand()` by issuing successive password resets and trying to reverse-engineer the internal state of `rand()`, or by pre-computing all 2^{32} possible results for `str_shuffle()`, given the `PASSWORD_CHARS` and `PASSWORD_LENGTH` environment variables.

Mitigation Strategies

Since libsodium is installed on the server, we propose the following be used instead:

```
function generate_password() {
    $random = base64_encode(
        Sodium::randombytes_buf(env('PASSWORD_LENGTH'))
    );
    $chars = rtrim(strtr($random, '+/', '-_'), '=');
    return substr($chars, 0, env('PASSWORD_LENGTH'));
}
```

Since libsodium employs a cryptographically secure pseudorandom number generator, this new function will return passwords with 84 bits of entropy.

$\log_2(64^{14})$ = exactly 84 bits

If 84 bits of entropy is not sufficient, then increase `PASSWORD_LENGTH` to meet the desired threshold of unpredictability.

Minimum Entropy Desired (bits)	`PASSWORD_LENGTH` (characters)
84 (current)	14 (current)
96	16
112	19
128	22
256	43

Resolution

This issue was promptly fixed after the initial version of this report was delivered.

2. Two Cryptographic Weaknesses in Yubikey Library

Severity: **Low, Low**
Exploitability: **Difficult, Difficult**

Affects:

- `vendor/bitbeans/yubikey/src/Yubikey.php`

Bytejail's two-factor authentication feature depends on the `bitbeans/Yubikey` library, which is a Laravel 5 compatible package for interfacing with the Yubikey API.

We identified two cryptographic weaknesses in this code:

1. Cryptographically insecure random byte string used for nonce
2. Non-timing-safe HMAC comparison

Despite the presence of both flaws, we consider the impact to be low, since the library has to initiate contact with the Yubikey API server for the methods that contained these issues to be invoked.

If an attacker is in a position to send forged Yubikey responses, they have already defeated the two-factor authentication (and possibly even compromised Yubico).

Mitigation Strategies

Our Chief Development Officer submitted a pull request on Github to expose and utilize a CSPRNG for nonce generation and to compare the HMAC signature in constant-time.

https://github.com/bitbeans/Yubikey/pull/1

This pull request was merged before the audit was complete.

IV. Other Findings

Note: The findings in this section are not necessarily vulnerabilities.

1. Sanitize All Output

Example:

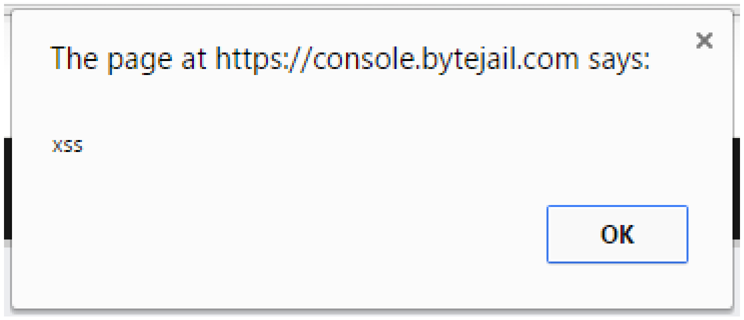
- `app/Jail.php`

The `getConvertedSize()` method returns an HTML string, like so:

```
return '<span id="'.$this->name.'-size" class="text-muted">'.$size.' '.
$suffix.'</span>';
```

We did not locate any point where an attacker could save a stored XSS to the jail name (or else this would have appeared in the Issues section). However, since it's not sanitized, setting the name field in the database to a specially crafted value does result in an alert box:

```
UPDATE jails SET name = '" onload=\'alert("xss");\' class="' WHERE id = 4;
```



The page at https://console.bytejail.com says:

xss

Attackers should not be in a position to edit arbitrary database values. Nonetheless, escaping any parameters that are placed directly in HTML attributes is a good practice when building a web application. (Ref. 2)

For example:

```
return '<span id="'.\htmlentities($this->name, ENT_QUOTES | ENT_HTML5 |
ENT_SUBSTITUTE, 'UTF-8').'-size" class="text-muted">'.$size.' '.$suffix.'</span>';
```

It may be helpful to define and utilize a helper function for this purpose.

Resolution

This was promptly fixed after the initial version of this report was delivered.

2. Using {!! $var !!} in Blade Templates Might be Dangerous

Similar to the previous finding, there were several places where storing HTML attribute escapes in the database led to a stored cross-site scripting condition.

This was promptly fixed after the initial version of this report was delivered.

3. For enhanced user privacy, add a daily nonce to IP obfuscation

When logging IPv4 addresses in the `LoginAttempt` model, the remote user's IPv4 address is hashed with the BLAKE2b hash function. From a user privacy perspective, this is a great idea. However, brute-forcing all possible IPv4 addresses and their respective BLAKE2b hash is not a difficult task.

A simple and effective way to enhance this privacy protection is to store a key that lives for only 24 hours in shared memory or in a temporary directory and overwrite it when the calendar changes. This enables anonymous user analytics with a degree of forward secrecy.

Resolution

This was promptly implemented after the initial version of this report was delivered.

4. Password hashing strategy might be controversial

Passwords are authenticated with (in order):

- BLAKE2b (client-side; provided by libsodium)
- Whirlpool
- SHA512
- Scrypt (Salsa20/8 + SHA256; provided by libsodium)

Some information security professionals advise against cascading hash functions. (Ref. 3)

The Whirlpool hash is invoked not only on the BLAKE2b hash; a pepper is appended as well. Peppers don't really offer more security than a salt alone; additionally, in the event of a compromise they offer no agility. The only solution would be to generate a new pepper and perform a unilateral password reset. (Ref. 4)

A more robust alternative to peppering the hashes would be to encrypt them after they are hashed, then decrypt them before verifying the password. We have previously published a library that hashes passwords with bcrypt then encrypts them with authenticated symmetric encryption (AES-CBC + HMAC-SHA-256 with PKCS#7 padding). (Ref. 5)

As it so happens, libsodium offers `Sodium::crypto_secretbox()` and `Sodium::crypto_secretbox_open()` for robust symmetric-key encryption. For stateful communications, `Sodium::crypto_aead_chacha20poly1305_encrypt()` and `Sodium::crypto_aead_chacha20poly1305_decrypt()` are available in newer versions of libsodium.

We recommend encryption over peppering; although there are no known security vulnerabilities with the existing construction.

Resolution

Shortly after our initial audit report, their password strategy was changed from peppering to employ encryption.

5. Minor Code Duplication

We spotted several places where an email was being sent and that particular method had to decide whether or not to encrypt it with the recipient's GnuPG public key. While none of the implementations were buggy or significantly different from the others, abstracting this feature into its own class or trait will make it easier for your team to maintain.

Resolution

This was remediated after the initial version of this report was delivered.

6. Negative Findings

We did not find any SQL injection vulnerabilities, nor even any non-vulnerable instances of string concatenation in a SQL query. This is a sign of quality security engineering.

We originally suspected that the use of `strlen()` in the Yubikey validation logic might fall prey to `mbstring.func_overload` bugs, but only ASCII-compatible strings were ever operated on, so we don't believe this will be an issue.

The `validate_bytejail_identity()` helper contains a truncated SHA-256 checksum which is susceptible to both brute-forcing (it's only 4 bytes long) and timing attacks (compared with `===` rather than `hash_equals()` or `Sodium::sodium_memcmp()`). Since it's only a checksum and not a MAC, this is entirely permissible and not security affecting at all.

A member of the Paragon Initiative Enterprises team had previously disclosed a corner-case security vulnerability in all recent versions of the Laravel framework. This vulnerability did not affect Bytejail at all, as it is configured to store session data in files, rather than cookies, and encryption is turned on.

The server is configured to only communicate over TLS 1.2 with perfect forward secrecy. Hypertext Strict Transport Security is properly used, port 80 only redirects to port 443, gzip compression is disabled on the `console.bytejail.com` server, and a HPKP header is sent to clients to mitigate a certificate authority compromise turned man-in-the-middle attack. In short, the webserver configuration for HTTPS is incredibly resilient and proactively secures clients from malicious third parties.

A Content-Security-Policy header is provided by a Laravel middleware class, to limit the potential impact of possible future XSS vulnerabilities. This is unambiguously a good thing.

V. Conclusion

Apart from a handful of easily addressable issues and noncritical findings, we found the Bytejail console to be robustly designed. Its reliance on libsodium for nearly all of its cryptographic operations significantly reduced the defect rate to nearly zero.

Our follow-up investigation found that all of the issues we reported were fixed, and furthermore our other findings were acted on.

Since the Bytejail development team adopted a strict approach to output sanitation and fixed the password reset issue. The application should withstand future audits and attackers (barring vulnerabilities in Laravel or the PHP interpreter).

VI. References

1. https://bit.ly/1IAdvxm
2. https://defuse.ca/web-application-security.htm
3. http://security.stackexchange.com/a/19877
4. http://blog.ircmaxell.com/2015/03/security-issue-combining-bcrypt-with.html
5. https://github.com/paragonie/password_lock