THOR Tool: XCS Attack Mitigation

Charles Melis, Jason Scott
Illinois Institute of Technology, Chicago, IL, USA
cmelis@hawk.iit.edu, jscott30@hawk.iit.edu

***Abstract*—Our report on cross channel scripting (XCS) examines the security risks and potential attacks resulting from the integration of different communication channels and data transfer protocols in embedded web servers. We present techniques used by attackers to inject malicious code into a machine via non-web channels like SNMP and FTP. We suggest a tool named THOR in order to mitigate such attacks and prevent other kinds of attacks.**

***Keywords*—cross channel scripting (XCS), security risks, web applications, embedded web servers, non-web channels, SNMP or FTP, attacks, vulnerabilities, THOR tool.**

I. Introduction

The increasing complexity of modern applications and devices has led to the integration of multiple communication channels, allowing data exchange between different components. However, this integration also creates potential security risks that attackers can exploit to inject malware through different non-web channels. This malware will then execute an arbitrary code that will lead to XSS and CSRF attacks.This technique is known as cross channel scripting (XCS).

Most of our everyday life devices, such as printers, routers or cameras have embedded web services that are only accessible through the internal network [1]. These web services are not tested enough and don't have the security requirements because they are not publicly available.

In this report, we investigate the security implications and potential attacks resulting from XCS through non-web channels into an embedded web server. We analyze the techniques used by attackers to realize an exploit through these channels and demonstrate the feasibility of XCS attacks. Furthermore, we propose best practices for mitigating XCS vulnerabilities and introduce our THOR tool as a practical solution for implementing these practices.

By providing a comprehensive analysis of XCS in embedded web servers, our report aims to inform developers and security professionals about the risks associated with XCS and how to protect against them with our tool.

II. Background

A. Background

The threat of attacks on modern applications is ever-present, with malicious actors constantly searching for new ways to exploit vulnerabilities. Two common types of attacks are cross-site scripting (XSS) [2] and cross-site request forgery (CSRF) [3], which have been well studied and have well-known defense mechanisms. However, a lesser-known but increasingly dangerous attack is cross channel scripting (XCS) [4]. This attack was presented by Hristo Bojinov, Elie Bursztein, and Dan Boneh in 2009. They present how we can inject a malicious executable through non-web channels to perform a XSS or a CSRF attack into the user browser.

XSS attacks typically involve injecting malicious code into a website, which then executes in the user's browser. Most of the time the code injected is in the form of a JavaScript code. The XSS vulnerabilities are divided into 3 parts [5]. The Stored XSS: This occurs when an attacker injects malicious code into a web page that is stored on the server and then served to all users who view that page. The Reflected XSS: This occurs when an attacker injects malicious code into a web page that is reflected back to the user in a response from the server. This type of attack is often used in phishing scams. The DOM-based XSS: This occurs when an attacker injects malicious code into a web page that is executed by the victim's browser when the page is loaded or interacted with. The final goal of the XCS attack is to perform a stored XSS attack in the user browser. While XSS can cause significant damage, its impact is typically limited to the context of the website in question but can lead to other attacks.

CSRF attacks involve tricking a user by doing malicious actions on his behalf. The attacker performs actions on a web application with the user privileges without the user consent or awareness.

On the other hand, XCS attacks affect a significant amount of embedded web servers that are usually not considered to be attacked. Consequently, these embedded web servers don't have the necessary mechanisms to prevent such attacks. Attackers can inject malware that aims to perform an XSS or CSRF attack through a non-web channel that will first exploit the vulnerability of the embedded web service. The attacker usually creates the JavaScript code so it can connect to one of his malicious websites to leverage his privileges on the victim's machine.

In this report, we investigate the security implications of XCS in embedded web servers. We analyze the techniques used by attackers to exploit these channels and demonstrate the feasibility of XCS attacks through a series of experiments. Furthermore, we propose best practices for mitigating XCS vulnerabilities and introduce our THOR tool as a practical solution for mitigating these attacks. Our findings highlight the need for increased awareness and proactive measures to protect against XCS attacks in modern devices.

The XCS attack has 3 steps to go through as explained in Figure 1 below (this image comes from [4]). The first step is the injection of the malware into the device. The device stores the malware, the malware has two possibilities. The first one to actively check when the user connects to the web server to monitor the device and then perform the XSS attack. The second one is to directly modify the HTML file to directly load the script when the user connects with his browser. Most of the time the user won't even know that he is being hacked because the actions are performed in the background.

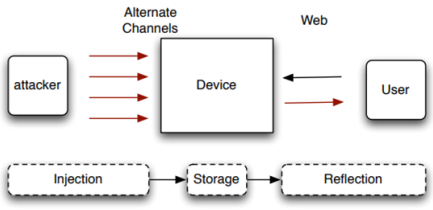

Figure 1: Overview of the XCS attack.

We can see in Figure 2 the scenario of XCS payload execution and various attack types [6]. We can see that one attack type targets confidentiality. This type of attack is called RXCS (Reverse Cross Channel Scripting), it consists in using a web interface/program as a benchmark to attack a further service on the network device it is known as reverse cross channel scripting. RXCS attacks are mainly used for unauthorized copying, transfer, or retrieval of data that is protected by access control.

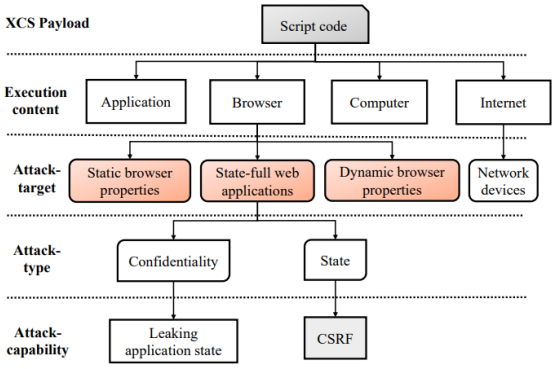

Figure 2: XCS payload execution and attack types.

B. Initial Research

Our initial research into XCS, a cross-channel scripting attack, has revealed that it is a particularly insidious form of attack, where a non-web channel like SNMP or FTP is used to inject a persistent XSS exploit that activates when the user connects to the web interface. This method of attack is more difficult to detect than traditional XSS attacks, as non-web channels are used to inject a malicious executable file in the device. Due to the huge number and specificities of non-web channels, we can not propose a defense mechanism based on them.

Furthermore, we found that the security of embedded web servers is often underestimated and not checked enough, leaving them vulnerable to XCS attacks. The consequences of XCS attacks can be severe, including the exfiltration of sensitive data like NAS-protected files or a user's keystrokes. XCS attacks can also redirect the user to a drive-by-download site or a phishing site, or exploit the user's device for DDoS attacks or for proxying the attacker's traffic as shown in Figure 3.

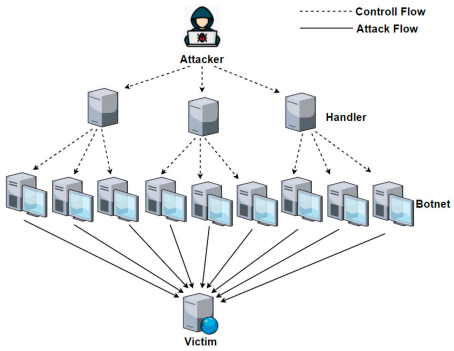

Figure 3: Distributed Denial Of Service attack proxied.

Given these findings, it is clear that XCS attacks pose a significant threat to modern applications, and that proactive measures must be taken to prevent them. Our report provides valuable insights into XCS attacks and suggests best practices for mitigating vulnerabilities, including the use of our THOR tool. We describe our THOR tool in the next section.

III. The Thor tool

A. What we target

In the paper we read, all the defense mechanisms encountered were on the browser [4]. In the paper aforesaid, they created a SiteFirewall that acts as a whitelist. Each webpage of the embedded web service that is loaded comes with a cookie that specifies a list of web pages that it can access. This mitigation is a client side algorithm you can see in Figure 4 below.

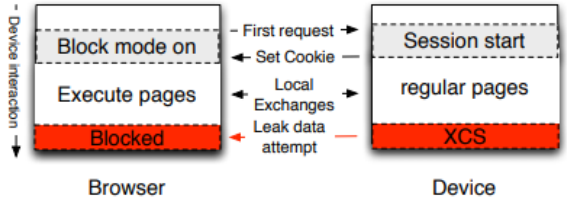

Figure 4: Interaction between the browser and embedded web site, with SiteFirewall enabled.

The main problem with the defense discussed above is that the malicious code can still be executed and the XSS attack can still take place. Our tool follows the recommendations of [6], we propose a server side tool to mitigate the payload that is originally implemented. Our tool is fully coded with python. As python is an easy language to understand we will show pictures of our code instead of a pseudo code that would be almost the same as the already existing python code.

Our tool is separated into multiple categories, each one of them is created to target a particular kind of analysis. This tool of malware mitigation is not only focused on XSS or CSRF attacks, but it focuses on malware in generales. Since devices that contain embedded web services can download and execute arbitrary code. Our tool can also be used to forbid malware such as ransomware to perform and to spread. As well as a trojan to connect to another device on the network that the attacker has access to and can steal some sensitive information. Our tool is used to monitor the current folder in which it is running and perform different kinds of analysis of a new executable file that is injected. The monitoring of the file is used to check the executable that has been inserted no matter what protocol has been used to inject it. The monitoring is done through watchdog.events and watchdog.observers python libraries.

The first analysis step is the pre-static analysis. This uses an API call to virus total in order to verify the hash sha-256 of an executable file. The second kind of analysis is the static analysis, our code extracts features from the debugger of the .exe file and checks if there are some that can be classified as malicious. The third one is a dynamic analysis, our code executes the executable in a special thread and checks if there are dangerous API calls. The last check is a machine learning analysis with a random forest. Each kind of analysis will be presented and discussed in the sections below.

B. Pre-Static analysis

As discussed before, our script monitors executables that have just been injected into the current folder. The pre-static analysis is based on Virus Totale. First, it takes the file in parameter , it parses the content of the file and creates a hash sha-256. Then, it connects through virus total via their API, and parses the JSON results. It checks the number of antivirus that detects it and gives the severity depending on this number. The code of the definition is shown in figure 5.

```
56  def check_virustotal(file_path):
57      with open(file_path, 'rb') as f:
58          file_content = f.read()
59      file_hash = hashlib.sha256(file_content).hexdigest()
60
61      url = 'https://www.virustotal.com/vtapi/v2/file/report'
62      params = {'apikey': API_KEY, 'resource': file_hash}
63      response = requests.get(url, params=params)
64      if response.status_code == 200:
65          json_response = response.json()
66          if json_response['response_code'] == 1:
67              positives = json_response['positives']
68              total = json_response['total']
69              if positives > 0:
70                  severity = get_severity(json_response)
71                  return True, severity
72      return False, None
```

Figure 5: Pre-static analysis through virus total

I based my code on the number of detections because I wanted to avoid false positives. If a lot of antiviruses classify it as a malware, it must be because it is one. You can see the function `get_severity` in figure 6.

```
def get_severity(json_response):
    engines = json_response['scans']
    detections = [engine for engine in engines if engines[engine]['detected']]
    num_detections = len(detections)
    if num_detections == 0:
        return 'Unknown'
    elif num_detections < 3:
        return 'Low'
    elif num_detections < 10:
        return 'Medium'
    else:
        return 'High'
```

Figure 6: Get the severity via the pre-static analysis

C. *The static analysis*

The main thing about this analysis is that it extracts features from the new file that has just been inserted. So it loads an executable file thanks to the PE library. It reads the file header and the section data and provides access to various attributes of the files. Firstly, it checks if the file is a DLL or EXE thanks to examination of the IMAGE_FILE_DLL or IMAGE_FILE_EXECUTABLE_IMAGE flag of the header. Then it checks whether there is a debug directory that most malwares have. Malware often has a debug directory that contains information about the compiler and linker used to create the executable. It checks the DIRECTORY_ENTRY_DEBUG attribute of the PE object. The code then creates a loop to check if there are functions imported from 'kernel32.dl' such as 'CreateProcess', 'WriteMemory', and 'VirtualAlloc' that are often used for malwares. These functions are often used by malware to create new processes, write code to memory, and allocate memory to run malicious code. If any of these functions are found in the import table, the function returns True, indicating that the file is a malware.The second loop iterates over the resources in the directory and checks if any of them have names that contain words such as 'config', 'setup', or 'install'. These words are often used by malware to create installation and configuration files. It also checks if they try to import malicious content such as string and images. The static analysis code is shown in figure 7.

```python
def static_malware_check(file_path):
    # Load the PE file
    try:
        pe = pefile.PE(file_path)
    except pefile.PEFormatError:
        # The file is not a valid PE file
        return False

    # Check if the file is marked as a DLL or an EXE
    if (pe.FILE_HEADER.Characteristics & 0x2000) or \
        (pe.FILE_HEADER.Characteristics & 0x0002):
        # Check if the file has a debug directory
        if hasattr(pe, 'DIRECTORY_ENTRY_DEBUG'):
            return True
        # Check if the file imports any suspicious APIs
        for entry in pe.DIRECTORY_ENTRY_IMPORT:
            if 'kernel32.dll' in entry.dll.lower():
                for imp in entry.imports:
                    if imp.name:
                        api_name = imp.name.lower()
                        if 'createprocess' in api_name or \
                           'writememory' in api_name or \
                           'virtualalloc' in api_name:
                            return True
        # Check if the file contains suspicious resources
        for resource_type in pe.DIRECTORY_ENTRY_RESOURCE.entrie
            if resource_type.name:
                if 'string' in resource_type.name.lower():
                    return True
            for resource in resource_type.directory.entries:
                if resource.name:
                    if 'config' in resource.name.lower() or \
                       'setup' in resource.name.lower() or \
                       'install' in resource.name.lower():
                        return True
    return False
```

Figure 7: Static analysis feature check

In the code above we can see the extension of the feature through the debugger of the file. Then in the main menu we check if the static analysis returns true. If yes we remove the file.

D. *Dynamic analysis*

In the previous sections we discussed the pre-static analysis and the static analysis, but these methods are not complete to have a high success rate. This is why we implemented the function of dynamic analysis in figure 8.

```python
# Define the main function to perform dynamic analysis
def dynamic_malware_check(file_path):
    # Execute the file and monitor its behavior
    process = subprocess.Popen([file_path])
    pid = process.pid
    p = psutil.Process(pid)
    mem = pymem.Pymem(pid)

    # Monitor process memory and API calls
    num_suspicious_activities = 0
    for _ in range(100):
        try:
            # Check for suspicious API calls
            for call in p.connections(kind='inet'):
                if call.status == 'ESTABLISHED':
                    num_suspicious_activities += 1
                    break

            # Check for suspicious memory activity
            if p.status() == psutil.STATUS_RUNNING:
                mem_data = mem.read_bytes(mem.process_base.lpBaseOfDll, mem.process_base.SizeOfImage)
                if b'CreateProcess' in mem_data or \
                   b'WriteProcessMemory' in mem_data or \
                   b'VirtualAlloc' in mem_data:
                    num_suspicious_activities += 1

            # Check for suspicious memory activity
            mem_data = mem.read_bytes(mem.process_base.lpBaseOfDll, mem.process_base.SizeOfImage)
            if b'CreateProcess' in mem_data or \
               b'WriteProcessMemory' in mem_data or \
               b'VirtualAlloc' in mem_data:
                num_suspicious_activities += 1

            # If multiple suspicious activities are detected, assume the file is malware
            if num_suspicious_activities >= 2:
                return "High Severity Malware"
        except psutil.NoSuchProcess:
            # The process has terminated
            break

    # If no suspicious activities were detected, assume the file is not malware
    return "Not Malware"
```

Figure 8: Dynamic analysis feature check

We firstly take the path of the file, then we take the file and execute it as a new process. As we create a new process we need to retrieve its process ID (PID). The function psutil.process (PID), creates an object for the process and it lets us monitor the behavior. The pymem function creates a pymem object in order to monitor the memory of the process. Then we keep a record of any malicious activity. So firstly we track if any TCP or UDP connection has been created thanks to the 'inet' parameter. If a connection is established, we increment the suspicious activity counter by 1. We try to see if any suspicious API is called by the process. Then, with the pymem, we try to see if there is any suspicious activity in memory such as CreateProcess, WriteProcessMemory or VirtualAllocation to write some code in the memory.

E. The machine learning analysis

The machine learning algorithm is another part of the project. We must not forget that our malware scanner will be implemented on the server side. The scanner must stay light to run on the machine without encountering any problems. We propose a machine learning algorithm that uses a RandomForestClassifier, and an ExtraTreesClassifier to train a model. Once the model trained with a data set “uci_malware_detection” find on Malware Executable Detection | Kaggle. We registered the models as model.pkl and features.pkl in a model folder. A Python pickle file serializes a tuple of two numpy arrays, (feature, label). There is no notion of "sentences" in pickle files; in other words, a pickle file stores exactly one sentence. feature is a 2-D numpy array, where each row is the feature vector of one instance; label is a 1-D numpy array, where each element is the class label of one instance. We use the model trained to detect whether a file monitored is a malware or not. The problem with this method is that we need to have the model on the server. These models take some memory space and must be available for the classification. If one of them is missing the compilation won’t be possible. Therefore, the machine learning algorithm is proposed to be added on the devices that have enough resources to run it.

```
def monitor_folder(folder):
    event_handler = NewFileHandler()
    observer = Observer()
    observer.schedule(event_handler, folder, recursive=False)
    observer.start()
    print(f'Monitoring folder {folder} for new file additions...')
    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        observer.stop()
    observer.join()
```

Figure 9: Monitoring the folder for the machine learning analysis

The function above monitors if a file is inserted into the current folder. It is exactly the same as the scanner with pre-static, static and dynamic analysis.

```
def checkFile(file):
    model = joblib.load("model/model.pkl")
    features = pickle.loads(open(os.path.join('model/features.pkl'), 'rb').read())
    data = extract_info(file)
    if data != {}:
        pe_features = list(map(lambda x: data[x], features))
        res = model.predict([pe_features])[0]
    else:
        res = 1
    return res
```

Figure 10: Check the file in the folder

This function first loads the train model with joblib and pickle. Then we see if this match the The function calls an extract_info function to extract information from the file. The extracted information is stored in the data variable.If the data variable is not empty, the function extracts specific features from the data dictionary using a lambda function and the map function. The extracted features are stored in the pe_features variable. The function then uses the loaded model to predict a result based on the extracted features. The predicted result is stored in the res variable.

We have a lot of functions in this code so I will explain them generally without showing the 222 lines of code.

So we have the functions : get_entropy,get_resources,get_version_info,extract_info.

1. *get_entropy*

The get_entropy function calculates the entropy of a file. Entropy is a measure of randomness in a data set. The function takes an input data set and calculates the frequency of each byte value in the data set. It then calculates the entropy by using the frequency of each byte value to determine the probability of that value occurring, and then takes the negative logarithm of that probability with base 2. The entropy is then calculated by summing the entropy values for each byte value.

2. *get_resources*

The get_resources function is used to extract the resources from a PE file. Resources can include images, icons, sound files, and other data that is embedded in the file. The function takes a PE file as input and returns a list of tuples that contain the entropy and size of each resource in the file.

3. *get_version_info*

The get_version_info function is used to extract the version information from a PE file. Version information can include the product name, company name, file description, file version, product version, and other information about the file. The function takes a PE file as input and returns a dictionary containing the version information.

4. *extract_info*

The extract_info function is used to extract information from a PE file. The function takes a file path as input, opens the file using the pefile module, and then extracts various properties of the file, such as the machine type, file size, and entry point. It then returns a dictionary containing this information.

With the functions described above we have multiple ways to check whether a file is a malware or not. And

this even without limiting ourselves to the exploit of a CSRF or XSS. In the next section, I'll present our real-world lab environment.

IV. The Lab Environment

A. *Lab simulation*

We first try to reproduce the laboratory environment. First of all, we have thanks to virtualbox install a Windows11 virtual machine in order to manipulate the malwares without any risk of being infected. Plus we have a copy on a GitLab of every file in case there is a problem with the manipulation of those malware.

To reproduce a real world environment we must create/emulate a device that contains an embedded web service and can accept executable files in its folder. For this purpose we use a google chrome extension named Web Server for Chrome [11]. This extension permits us to have a web server accessible in our internal network. An example of the web service of the device is shown in Figure 11.

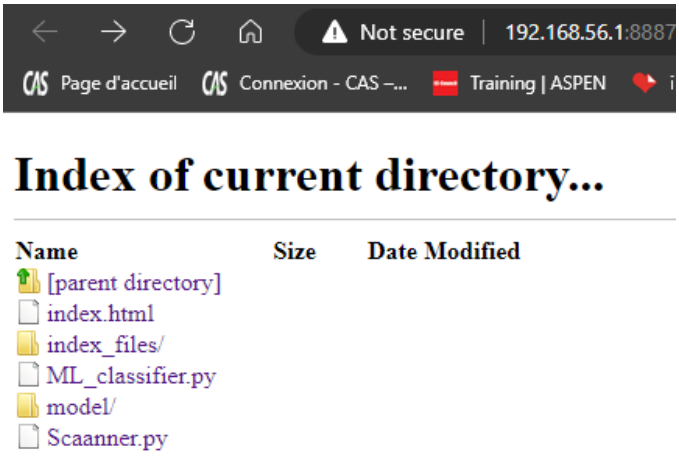

Figure 11: Device web service simulation

When we connect directly to the internal address, The login page of my router is shown by default to simulate what a normal user will view if he connects to the device. We can see that our files will monitor the current folder where the index.html is.

B. *Adversary mitigation*

To present correctly our defense mechanisms we must introduce the malware that we've tested. We used malware available in well known database centers such as MalwareBazaar [12] and Virusshare [13].

MalwareBazaar is a popular website that provides a free and open repository of malware samples. You can download malware samples and try our defense mechanism.

Virusshare is a popular website that provides a free malware sample sharing service. It contains a large collection of malware samples that you can download and try your defense mechanisms with.

The advantage of these two websites is that you can download really recent malwares and have a ton of information about it. An example is presented in Figure 12.

SHA256 hash:	4be412cab845dfa0a80431b758bf2196708522eb4d6d1
SHA3-384 hash:	4ba07360ebccbb669e610e181c410dfa847f7d6b1e19a
SHA1 hash:	2b09468f4ddf52f9c40c4a9ecd47a0460e03f7de
MD5 hash:	0276be39190d11708dd28c908575086a
humanhash:	william-lion-mirror-hawaii
File name:	M100538944177.exe
Download:	download sample
Signature	Loki Alert
File size:	489'472 bytes
First seen:	2023-04-24 04:50:11 UTC
Last seen:	*Never*
File type:	exe
MIME type:	application/x-dosexec
imphash	f34d5f2d4577ed6d9ceec516c1f5a744 (36'494 x Agent
ssdeep	12288:/J0PfaUPJE8odDmAh8r62DumhlBp+9P5MltI:yPf
Threatray	4'056 similar samples on MalwareBazaar
TLSH	T180A402942277B5EADDCC17BB4660245D037061832
TrID	63.0% (.EXE) Generic CIL Executable (.NET, Mono, etc.) (7 11.2% (.SCR) Windows screen saver (13097/50/3) 9.0% (.EXE) Win64 Executable (generic) (10523/12/4) 5.6% (.DLL) Win32 Dynamic Link Library (generic) (6578/2 3.8% (.EXE) Win32 Executable (generic) (4505/5/1)

Figure 12: Example of information of a malware in MalwareBazaare

In the figure above, we can see the hash, the file type, the signature, the date. On MalwareBazaar we can also see the IOCs (indices of compromission) of a file. We used the two sites cited before in order to perform a test with up-to-date malware that is not always already discovered.

We have tested our defense mechanism against 31 malwares in order to see the efficiency of our mechanism. All the malware chosen are executable files. We will present our results of these tests in the next section.

V. Evaluation

A. *The overall results*

To present our results we present overall detection when the scanner and the machine learning are active. The first graph is presented in Figure 13.

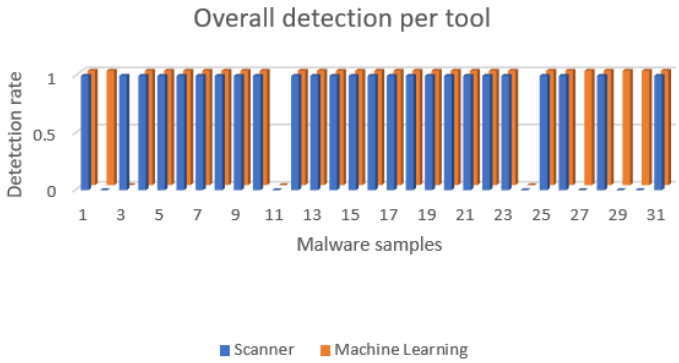

Figure 13: Detection with our 2 modules

We can see in the histogram above, that our model has been well trained. Each bar at 1 is a detection, each bar at 0 is not a detection. The Scanner has an average of 86,3% detection rate. Meanwhile, the machine learning model has a 90% detection rate with a precision rate of 80%. The deficiency detection of the scanner can be explained by the lack of suspicious API check. The first time the result of the scanner was not satisfactory so we added some suspicious APIs to check. After this modification we had a satisfactory detection rate. With these results we investigate which one has a most probable detection rate. We study the detection rate with each method in the section below.

B. The results by mechanism

We saw in the section above the global detection rates. We will then investigate in detail the mechanisms. The Figure 14, shows the details of each mechanism used in the scaanner.py file.

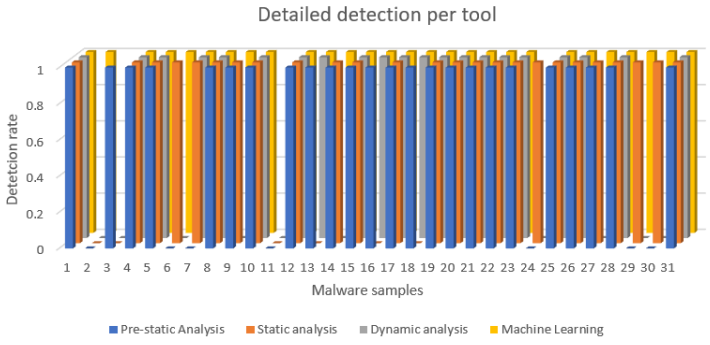

Figure 14: Detection detailed by each mechanism of the tool

Here we see the detail of the rate detection. We can see that the pre-static analysis has a detection rate of 83,2 %. The static analysis shows a detection rate of 79,7% and the dynamic analysis has a detection rate of 82,4%. The lower detection rate of the scanner is explained by a lack of significant API list. But the overall result is still acceptable.

When we combine our 2 tools together, we have a 91% detection rate of the malware detection. This result is acceptable because the device must have a low power consumption due to the fact that the threat is on an embedded server. But our result is still inferior to papers like [14], [15], [16]. All the papers cited previously are either IOT detection or lightweight malware detection.

VI. Conclusion

In this report, we designed a tool named THOR tool. This tool aims to prevent and mitigate XCS (Cross Channel Scripting) attacks as well as other attacks that could exploit a device by injecting malware through a non web channel such as FTP or SMB.

Our tool has two main categories, the first one is a scanner. The scanner contains 3 types of analysis, the pre-static, the static analysis and the dynamic analysis. The pre-static analysis connects to virus total to see if the executable file is a virus. The static analysis extracts features from the file and checks if they are malicious or not. The dynamic analysis monitors the behavior of a file and checks if there is any malicious API call. The machine learning malware analysis loads a pretrained model from a dataset to classify an executable file as a malware or not.

We checked the behavior of our tools in a realistic and secure environment. The test has been realized with some recent malware in order to have an accurate and up-to-date result.

We saw that our overall results are satisfactory when both tools are combined. But we also saw that these tools have some energy consumption and must stay lightweight. We showed that we can combine all of our tools together or to run either the scanner or the machine learning algorithm in order to mitigate the threat.

We think that some options should be considered for future research is to implement a similar tool that can be more lightweight. Another research subject should be to realise a survey among constructor to implement this tool and see the drawbacks in real life devices. Plus to see if the risk is worth the price to implement such a similar tool. Researchers can also try to implement security protocols for all of the devices that have embedded web services. They can implement a new secure data transfer protocol for these devices that can check whether the data passed through this device is the data that should transit.

The repository of our tool has been imported from GitLab to Github. The Link to access the repository is : [noiuytre/CS-528-XCS-attack-mitigation (github.com)]

References

[1] Q. Kang, H. He and H. Wang, "Study on Embedded Web Server and Realization," 2006 First International Symposium on Pervasive Computing and Applications, Urumqi, China, 2006, pp. 675-678, doi: 10.1109/SPCA.2006.297507.

[2] Song, Xuyan & Zhang, Ruxian & Dong, Qingqing & Cui, Baojiang. (2023). Grey-Box Fuzzing Based on Reinforcement Learning for XSS Vulnerabilities. Applied Sciences. 13. 2482. 10.3390/app13042482. .

[3] C. Liu, X. Shen, M. Gao and W. Dai, "CSRF Detection Based on Graph Data Mining," 2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE), Dalian, China, 2020, pp. 475-480, doi: 10.1109/ICISCAE51034.2020.9236806.

[4] Hristo Bojinov, Elie Bursztein, and Dan Boneh. 2009. XCS: cross channel scripting and its impact on web applications. In Proceedings of the 16th ACM conference on Computer and communications security (CCS '09). Association for Computing Machinery, New York, NY, USA, 420–431.

[5] K. Santithanmanan, "The Detection Method for XSS Attacks on NFV by Using Machine Learning Models," 2022 International Conference on Decision Aid Sciences and Applications (DASA), Chiangrai, Thailand, 2022, pp. 620-623, doi: 10.1109/DASA54658.2022.9765122.

[6] M, I.; Kaur, M.; Raj, M.; R, S.; Lee, H.-N. Cross Channel Scripting and Code Injection Attacks on Web and Cloud-Based Applications: A Comprehensive Review. Sensors 2022, 22, 1959.

[7] Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G. Saner: Composing static and dynamic analysis to validate sanitization in Web applications. In IEEE Symposium on Security and Privacy (2008)

[8] B. Bencsáth, L. Buttyán and T. Paulik, "XCS based hidden firmware modification on embedded devices," SoftCOM 2011, 19th International Conference on Software, Telecommunications and Computer Networks, Split, Croatia, 2011, pp. 1-5.

[9] sklearn.ensemble.RandomForestClassifier — scikit-learn 1.2.2 documentation

[10] ML | Extra Tree Classifier for Feature Selection - GeeksforGeeks

[11] https://chrome.google.com/webstore/detail/web-server-for-chrome/ofhbbkphhbklhfoeikjpcbhemlocgigb?hl=en

[12] MalwareBazaar | Browse malware samples (abuse.ch)

[13] VirusShare.com

[14] Marjan Golmaryami, Rahim Taheri, Zahra Pooranian, Mohammad Shojafar, and Pei Xiao. 2022. SETTI: A Self-supervised AdvErsarial Malware DeTection ArchiTecture in an IoT Environment. ACM Trans. Multimedia Comput. Commun. Appl. 18, 2s, Article 122 (June 2022), 21 pages. https://doi.org/10.1145/3536425

[15] Ahmed Abusnaina, Afsah Anwar, Sultan Alshamrani, Abdulrahman Alabduljabbar, RhongHo Jang, DaeHun Nyang, and David Mohaisen. 2022. Systematically Evaluating the Robustness of ML-based IoT Malware Detection Systems. In Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '22). Association for Computing Machinery, New York, NY, USA, 308–320. https://doi.org/10.1145/3545948.3545960

[16] D. -O. Won, Y. -N. Jang and S. -W. Lee, "PlausMal-GAN: Plausible Malware Training Based on Generative Adversarial Networks for Analogous Zero-Day Malware Detection," in IEEE Transactions on Emerging Topics in Computing, vol. 11, no. 1, pp. 82-94, 1 Jan.-March 2023, doi: 10.1109/TETC.2022.3170544.

[17] noiuytre/CS-528-XCS-attack-mitigation (github.com)