Annals of Emerging Technologies in Computing (AETiC)
Vol. 5, No. 1, 2021

Research Article

Detection of the Hardcoded Login Information from Socket and String Compare Symbols

Minami Yoda[1,*], Shuji Sakuraba[1], Yuichi Sei[1,2], Yasuyuki Tahara[1] and Akihiko Ohsuga[1]

[1]The University of Electro-Communications, Tokyo, Japan
yoda.minami@ohsuga.lab.uec.ac.jp; tahara@uec.ac.jp; ohsuga@uec.ac.jp; sakuraba.shuji@ohsuga.is.uec.ac.jp
[2]JST PRESTO, Saitama, Japan
seiuny@uec.ac.jp
*Correspondence: yoda.minami@ohsuga.lab.uec.ac.jp

Received: 8th November 2020; Accepted: 23rd December 2020; Published: 1st January 2021

Abstract: Internet of Things (IoT) for smart homes enhances convenience; however, it also introduces the risk of the leakage of private data. TOP10 IoT of OWASP 2018 shows that the first vulnerability is "Weak, easy to predict, or embedded passwords." This problem poses a risk because a user can not fix, change, or detect a password if it is embedded in firmware because only the developer of the firmware can control an update. In this study, we propose a lightweight method to detect the hardcoded username and password in IoT devices using a static analysis called Socket Search and String Search to protect from first vulnerability from 2018 OWASP TOP 10 for the IoT device. The hardcoded login information can be obtained by comparing the user input with strcmp or strncmp. Previous studies analyzed the symbols of strcmp or strncmp to detect the hardcoded login information. However, those studies required a lot of time because of the usage of complicated algorithms such as symbolic execution. To develop a lightweight algorithm, we focus on a network function, such as the socket symbol in firmware, because the IoT device is compromised when it is invaded by someone via the Internet. We propose two methods to detect the hardcoded login information: string search and socket search. In string search, the algorithm finds a function that uses the strcmp or strncmp symbol. In socket search, the algorithm finds a function that is referenced by the socket symbol. In this experiment, we measured the ability of our proposed method by searching six firmware in the real world that has a backdoor. We ran three methods: string search, socket search, and whole search to compare the two methods. As a result, all methods found login information from five of six firmware and one unexpected password. Our method reduces the analysis time. The whole search generally takes 38 mins to complete, but our methods finish the search in 4-6 min.

Keywords: ***Backdoor; Internet of Things; Smart Home; Static Analysis***

---

1. Introduction

Smart speakers and smart home controllers have become popular. The Internet of Things (IoT) device provides services that improve our daily lives and some of these services are provided through the Internet. The average annual growth rate of IoT device market from 2018 to 2022 is predicted to be 20% [1]. However, attacks on IoT devices have also increased. According to F-Secure's report, the number of attacks on IoT devices in the first half of 2018 was 231 million, whereas that in

[1] IDC. All categories of smart home devices forecast to deliver double- digit growth through 2022, 2018. https://www.idc.com/tracker/showproductinfo.jsp?prod_id=1781

Minami Yoda, Shuji Sakuraba, Yuichi Sei, Yasuyuki Tahara and Akihiko Ohsuga, "Detection of the Hardcoded Login Information from Socket and String Compare Symbols", *Annals of Emerging Technologies in Computing (AETiC)*, Print ISSN: 2516-0281, Online ISSN: 2516-029X, pp. 28-39, Vol. 5, No. 1, 1st January 2021, Published by International Association of Educators and Researchers (IAER), DOI: 10.33166/AETiC.2021.01.003, Available: http://aetic.theiaer.org/archive/v5/v5n1/p3.html.

the first half of 2019 was 2.9 billion, denoting an increase by approximately 12 times [2]. The reported attacks included an attack to leak user privacy data. For example, a baby monitor was attacked in 2018, and a video of a baby was leaked. Furthermore, an attacker talked to the baby. If an outsider can view the occurrences inside our house, we will feel uncomfortable. Therefore, knowledge about IoT security is crucial to live safely with IoT devices [1], [2].

The OWASP TOP 10 classifies the most important categories of control and control that every architect and developer should include in their project on the basis of real-world vulnerabilities. Table 1 shows the ranking list of 2018[3].

In this study, we propose two methods to detect the hardcoded login information by analyzing the firmware, i.e., string search and socket search. String search finds a function that references the strcmp or strncmp symbol, whereas socket search finds a function that has contact with the socket symbol within a certain range and references the strcmp or strncmp symbol. This method was also able to detect the first vulnerability listed in the 2018 OWASP TOP 10 for IoT devices.

During the experiment, we measured the ability of the two methods and conducted the whole search using real-world firmware. All searches found hardcoded login information as candidates; both search methods reduced the time required when compared to the time required for the entire search. Socket search could find hardcoded login information with minimum candidates.

The purpose of the study was to propose the following:

- Method to detect the first vulnerability according to the top 10 OWASP in 2018.
- Algorithm to shorten analysis time by characterizing the hardcoded login information.

2. Background

The term backdoor is defined in many studies. Thomas et al. defined the backdoor as follows: ”The function of the device, which is not visible to the user, can be determined by using authorized functions and information and inserted with the intention of weakening security features” [3].

In this paper, we consider that the hardcoded login information is also a backdoor. Also, hardcoded login information is sometimes written in the user manual, so it is very hard to declare ”invisible.” However, because hardcoded login information cannot be deleted or changed by the user, it will allow anyone to have access to the IoT devices. This could be a huge vulnerability.

As a backdoor example in the real world, Thomas *et al.* [4] found hardcoded login information and trigger path to log in the firmware of the Q-See DVR. For example, Figure 1 shows a real-world backdoor code in Q-See DVR. strcmp(username, "admin"), strcmp("603huanyuan", password) is embedded as login information. After entering the login information, as a privileged user, it will be redirected to the control panel.

Table 1. Top 10 OWASP IoT Vulnerabilities in 2018

No.	Vulnerability
1st	Weak, easy to predict, or embedded passwords
2nd	Insecure communications services
3rd	Insecure ecosystem interface
4th	Lack of a secure mechanism for software updates
5th	Use of insecure or compromised software components
6th	Inadequate privacy protection
7th	Insecure data transfer and storage
8th	Lack of device management such as support
9th	Insecure standard settings
10th	Inadequate physical hardening

```
if ((local_39 == false) && (iVar1 = strcmp(local_34,"admin"),
  iVar1 = strcmp(local_38,"6036huanyuan");
```

Figure 1. Real-World Backdoor Code in Q- See DVR

[2] F-Secure. Attack landscape h1 2019, 2019. https://www.f-secure.com/content/dam/press/de/media-library/reports/F-Secure-attack-landscape-h12020.pdf

[3] OWASP. Owasp-iot-top-10-2018, 2018. https://wiki.owasp.org/index.php/OWASP_Internet_of_Things_Project

3. Related Work

Zhang *et al.* [5] focused on the network packet pattern of backdoors and proposed a method to detect backdoors by detecting network packet matching the pattern. They developed a general algorithm for detecting interactive traffic based on packet size and timing features, and a set of protocol-specific algorithms that look for signatures that distinguish particular protocols.

They evaluated the algorithms on large traces of Internet access and found they performed well. Moreover some algorithms can be prefiltered using a stateless packet filter that increases performance at little or no loss of accuracy. However, it is difficult to prevent from unauthorized logins because it is necessary to capture the network packet before the login.

Shoshitaishvili *et al.* [6] proposed a method to detect backdoors by symbolic execution called Firmalice. It is a binary analysis framework to support the analysis of firmware running on embedded devices. It builds on top of a symbolic execution engine and techniques, such as program slicing, to increase its scalability.

Their method defines a security policy that describes the trigger for the program's privileged operations and the characteristics of its privileged operations. The tool considers the path as a backdoor if it finds the path flow that reaches a privilege operation during symbolic execution, but not in the security policy. Symbolic execution is proposed by James C. King [7], which is a means of analyzing a program to determine what inputs cause each part of a program to execute.

During their experiment, they analyzed the firmware in which the backdoor was embedded and checked that the backdoor can be found. They evaluated Firmalice on the firmware of three commercially available devices and were able to detect authentication bypass backdoors in two of them. This method takes to analyze the entire firmware for 12 min of 1.9 MB data of the firmware and approximately 11h of 7.2 MB data of the firmware by symbolic execution. Their method has a limitation that it is not able to a flaw that deviates from its policy.

Thomas *et al.* [8] proposed a method for detecting backdoors by a classifier using semi-supervised learning called HumIDIFy. Their method gathers symbol information and learns information from binaries to semi-supervised vector support machine learning to create a backdoor detection model, so that it is compared to the expected functionality profile that their method defines by hand for a range of applications.

To specify these profiles, they developed a domain-specific language called Binary Functionality Description Language (BFDL), which encodes the static analysis passes used to identify specific functionality traits of a binary. HumIDIFy achieves a classification accuracy of 96.45% with virtually zero false positives for the most common services. They experimented with the applicability of our techniques to a large-scale analysis by measuring performance on a large data set of firmware. From sampling that data set, their method identifies a number of binaries containing unexpected functionality, notably a backdoor in router firmware by Tenda. Their method is effective in finding a backdoor; however, the method takes time and effort to generate a model before backdoor detection.

Thomas *et al.* [4] proposed other methods called Stringer. strcmp() and strncmp() are often used to compare a user input to embedded password string. Thus, these methods weight functions that are popular to the backdoor and determine the functions with high weight as candidates for a backdoor, then labels each function 's basic blocks with the set of sequences of static data that must be matched against to reach them. Then using these sets, it assigns a score to each function, which measures the extent to which the function 's branching is influenced by static data. They demonstrated the effectiveness of the approach to lightweight analysis by running it on a data set of 2,451,532 binaries from 30 different COTS device vendors. Their result shows their techniques are effective by discovering three backdoors and recovering a proprietary command set, two of which previously undocumented.

Salwan *et al.* [9] proposed an open-source software allowing to evaluate the proposed approach against several forms of virtualization. It is an effective method to find a clew of backdoor function by analysing a value in memory and data. They present a generic approach based on

exploration, tainting, and recompilation of the symbolic path, allowing the recovery from a virtualized code of a devirtualized code that is semi- identical to and close in size.

Yoda *et al.* [10] proposed two methods to detect the hardcoded login information-string search and socket search. They focused on the string and network function, which are often used by a backdoor. In their experiment, a few backdoors were founded around a network function. On the other hand, they found that a backdoor function which has a hardcoded information always use strcmp() and strncmp() symbol. However, their accuracy of string search has room for improvement.

Ming *et al.* [11] proposed StraightTaint, a novel technique for completely decoupling dynamic taint analysis for offline symbolic taint analysis. It realizes lightweight logging and much lower online execution slowdown; meanwhile, previous approaches rely on complete runtime values or inputs. The results of tool performance show that StraightTaint can rival dynamic taint analysis at a similar level of precision, but with a much lower online execution slowdown and more exible functionalities. The experimental evidence indicates that StraightTaint can be applied to speed up various expost facto security applications with full-featured offline taint analysis.

Yakdan *et al.* [12] proposed REcompile, an efficient and extensible decompilation framework. REcompile to produce wellreadable decompiled code compare to previous work. The overall evaluation, using real programs and malware samples, shows that REcompile achieves a comparable and, in many cases, better performance than state-of-the-art decompilers. The method uses the static single assignment (SSA) form as its intermediate representation and performs three main classes of analysis. Data flow analysis removes machine-specific details from code and transforms it into a concise high-level form. Type analysis finds variable types based on how those variables are used in code. Control flow analysis identifies high-level control structures such as conditionals, loops, and switch statements.

David *et al.* [13] proposed a reverse engineering framework, which recovers a program from a few debug information. They presented a novel approach for predicting procedure names in stripped executables. The approach combines static analysis with neural models.

The main idea is to use static analysis to obtain increased representations of call sites; to encode the structure of these call sites using the control flow graph (CFG); and to generate a target name while attending these call sites. They used LSTM-based and transformer-based architectures to drive graph-based. Its evaluation shows that the models produce predictions that are difficult and time consuming for humans, while improving on existing methods by 28% and by 100% over state-of-the-art neural textual models that do not use any static analysis.

Garmany *et al.* [14] proposed a static analysis framework to find uninitialized variables in binary executables. Their prototype implementation is capable of detecting uninitialized memory errors in complex binaries such as web browsers and OS kernels, and we detected seven novel bugs. The methods to lift the binaries into a knowledge representation which builds the base for specifically crafted algorithms to detect uninitialized reads.

Stoenescu *et al.* [15] proposed a binary analysis framework based on symbolic execution with the distinguishing capability to execute stepwise forward and also backward through the execution tree. This helps to find a value in memory and dynamic flow analysis. It was developed internally at Bitdefender and code-named RIVER. The framework provides components for constraint solving, such as a taint engine, a dynamic symbolic execution engine, and integration with Z3.

Cesare *et al.* [16] proposed Bugwise, which is a system that performs bug detection on x86 binary-level programs. The system employs static analysis and the novel application of decompilation to make that analysis tractable. The method is able to detect a number of bug classes, including use-after- frees, double frees, and buffer overflows using environment variables. Its results found tens of bugs and vulnerabilities in Debian Linux, scanning the entire repository of that Linux distribution. Bugwise shows that traditional static analysis can be applied to binaries through the use of decompilation techniques. However, source code is not always available, as in the case of a black-box penetration test.

Alrabaee *et al.* [17] proposed a novel technique that extracts the semantics of binary code in terms of both data and control flow. They implement the system in a tool called BinGold and

evaluate it against thirty binary code applications. Its technique allows robust binary analysis because the extracted semantics of the binary code is generally immune from light obfuscation, refactoring, and varying the compilers or compilation settings. To realize robust analysis, they applied data-flow analysis to extract the semantic flow of the registers, which are then synthesized into a novel representation called the semantic flow graph (SFG). After the step, it extracts various properties, such as reflexive, symmetric, antisymmetric, and transitive relations, applied to the binary analysis. Its evaluation shows that BinGold successfully determines the similarity between binaries, yielding highly robust results against light obfuscation and refactoring. In addition, they found that BinGold has other abilities to find a binary code authorship attribution and the detection of clone components across program executables.

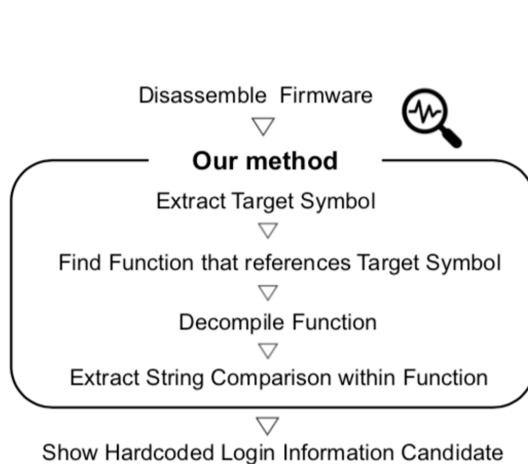

Figure 2. Overview of Approach

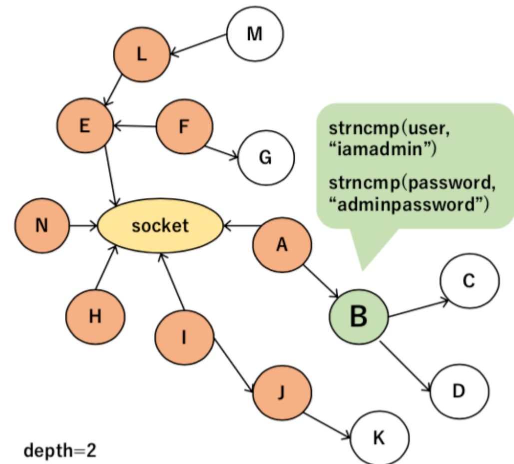

Figure 3. Socket Search

4. Approach

4.1. Overview

We propose a method that finds the line that uses strcmp or strncmp as a candidate of hardcoded login information, such as login ID and password in a firmware. Our method realizes a lightweight and short-time analysis by filtering a function that is related to a backdoor. We propose two detection ways: string search and socket search. In string searching, it finds a function that uses strcmp or strncmp symbol. In socket searching, it finds a function that is referenced by socket symbol.

Algorithm 1. Main Program of String Search

```
1:  SymbolTable ← getSymbolTable()
2:  while SymbolTable.hasNext() do
3:    Symbol ← SymbolTable.getSymbol()
4:    if Symbol.isMatched(strn?cmp) then
5:        SymAddress ← Symbol.getAddress()
6:        SymFunction ←getF unctionF romAddr(SymAddress)
7:        FunctionList ←getReferenceF unctions(SymFunction)
8:        for ChildFunction ∈ FunctionList do
9:          printHardCoded(ChildFunction)
10:       end for
11:   end if
12: end while
```

The difference between a string search and socket search is the filtering. String search extracts all functions that references a strcmp or strncmp symbol. In general, either strcmp or strncmp is used to compare the user input and hardcoded login information. Thus, it is effective to focus on searching these two symbols to find a backdoor.

Socket search is more focused on the network function. At related work, all backdoors were accessed via a TCP/IP connection, so there is a possibility that a backdoor function is located near a

network function. Socket symbol is used at network function, so we filter the function that references a socket symbol.

Algorithm 2. printHardCoded function

```
1: result ← getDecompiledF unction(Function)
2: lines ← result.eachLine.matches(". ∗ strn?cmp. ∗")).toList()
3: if lines.size() > 0 then
4:   lines.forEach(line− > println(line))
5: end if
```

Next, after finding the symbol, the method decompiles all functions within the functions list to extract strings. Finally, the tool extracts strings in a decompiled code and shows the line as hardcoded login information candidate.

Algorithm 3. Main Program of Socket Search

```
1: Depth ← 5
2: SymbolTable ← getSymbolTable()
3: while SymbolTable.hasNext() do
4:   Symbol ← SymbolTable.getSymbol()
5:   if Symbol.isSocket then
6:       SymAddress ← Symbol.getAddress()
7:       SymFunction ← getF unctionF romAddr(SymAddress)
8:       FunctionList ←getReferenceF unctions(SymFunction)
9:       for ChildFunction ∈ FunctionList do
10:          Incoming ← getIncomingCalls(ChildFunction)
11:          printReference(Incoming,Depth)
12:      end for
13:    end if
14: end while
```

Algorithm 4. printReference Function (Function, Depth)

```
1: FunctionList ← getReferenceF uncsF rom(Function)
2: for Function ∈ FunctionList do
3:   printHardCoded(Function)
4:   printIncomingCalls(Function,Depth)
5:   printOutgoingCalls(Function,Depth)
6: end for
```

Algorithm 5. printIncoming(Outgoing)Calls function (Child-Function, Depth)

```
1: FunctionList ← getReferenceF uncsF rom(Function)
2: for Function ∈ FunctionList do
3:   printHardCoded(Function)
4:   printIncomingCalls(Function,Depth)
5:   printOutgoingCalls(Function,Depth)
6: end for
```

4.2. String Search

String search finds a function and the line that uses a strcmp or strncmp symbol. These symbols are used to compare the user input and hardcoded login information. Thus, we think that the function that uses these symbols is a candidate for the backdoor function.

The main algorithm of string search is explained in Algorithm 1, and the related function is written in Algorithm 2. In the main program (Algorithm 1), we load a symbol table of firmware and check that a strcmp or strncmp symbol is in the symbol table (line 4). If the symbol is contained, we get the address of the strcmp or strncmp symbol to get a function information (lines 5–6). By using this function information, we make a list of function that the strcmp or strncmp function references (line 7). These functions in the list are labeled as a candidate of backdoor.

In the printHardCoded function in Algorithm 2, it decompiles the function that is passed as args. It decompiles each function in the list and picks the line that uses a strcmp or strncmp symbol string in the function. If the hardcoded string line is found, it shows the line as a result.

4.3. Socket Search

Socket search find the line that uses a strcmp or strncmp symbol around the socket function. In the related work, a backdoor is always accessible by the TCP/UDP function [6], so the hardcoded strings around the socket symbol could be a candidate of login information.

This method extracts the socket symbol and searches hardcoded strings around the symbol. Figure 3 shows the movement of socket searching. This is a call graph of the socket symbol. Each node means a function. An arrow means a reference relationship between the functions. For example, function A references the socket symbol. Socket is the standard function, so it is referenced by the other function. The search starts from the socket symbol. The cursor proceeds a function next to the socket. Depth means how far from the start point. When the cursor moves next to the function, a number of depths reduce one. The search continues until the depth is bigger than zero.

In this example, depth is two, so the search seeks for two hops from the start point. When a depth is two, orange and green nodes will be searched. In this case, function B has hardcoded strings, so the method shows the line of B.

We explain in more detail of this search in Algorithm 3. Our tool starts running the Main Program after reading the firmware by using the Ghidra software. Ghidra is a software reverse engineering (SRE) tool which is produced by the National Security Agency (NSA)[4] .

We set the Depth value to five (see line1) (depth means the number of step how far from the socket symbol). Then, it loads a symbol table of firmware and checks the socket symbol is in the symbol table (line 5). It gets the address of the socket symbol to get function information (lines 6–7). It extracts the functions that are referenced by the socket function (line 8). In the referenced function list, it gets an incoming function of each function (line 10). Then, it searches for a reference relationship of an incoming function with a depth parameter (line 11).

In the printReference function in Algorithm 4, it searches a reference of function and finds hardcoded strings in a function that is passed as args. In line 3, after decompiling, it tries finding the embedded ID and password. If it found a line that contains the strings, it shows the line as a result.

The printIncomingCalls function lists up all functions that are incoming from a parent function, and it reaches a reference by the depth at Algorithm 5. After listing up, it passes the function list to the printHardCoded function, and the tool shows the result. The printOutgoingCalls function works the same way that the printIncomingCalls do but the reference direction is the opposite. It searches for an outgoing function to a parent function.

Table 2. Real-World Hardcoded Login Information

Firmware Name	Login Information
D-Link Router	"xmlset_roodkcableoj28840ybtide"
Q-See DVR	strcmp("6036huanyuan",password)
Trendnet Router	"emptyuserrrrrrrrrrrr"
Tenda Router	strcmp("w302r_mfg",packet->magic)
TCP32764 Router	"ScMM"
Ray Sharp DVR	strcmp("519070",password) == 0

Table 3. Evaluation Index

	True Candidate	False Candidate
Method Found	True Positive (TP)	False Positive (FP)
Method Missed	False Negative (FN)	True Negative (TN)

$$Accuracy = \frac{TP+TN}{TP+FP+FN+TN} \quad (1)$$

$$Precision = \frac{TP}{TP+FP} \quad (2)$$

$$Recall = \frac{TP}{TP+FN} \quad (3)$$

$$F-Score = \frac{2 \times Precision \times Recall}{Precision+Recall} \quad (4)$$

[4] National Security Agency. Ghidra, 2019. https://www.ghidra-sre.org/.

```
int FUN_00060118(Backdoor)(int param_1,char *param_2,char *param_3)

{
  int iVar1;
  char *__s;
  int iVar2;
  int iVar3;
  int iVar4;

  FUN_000cef88(DAT_004a6788,*(undefined4 *)(param_1 + 0x334),0);
  iVar1 = strcmp(param_3,"664225");
  if ((iVar1 == 0) && (iVar1 = strcmp(param_2,"root"), iVar1 == 0)) {
    puts("LINE[1] REMOTE USER LOGIN IN!");
    return 1;
  }
```

Figure 4. Reported Login Password in Q-See DVR

```
else {
  iVar1 = strcmp(param_2,(char *)(*(int *)(DAT_004a6788 + 0x74) + 0x2c74));
  if (iVar1 != 0) {
    return 0xffffffff;
  }
  __s = "LINE[6] REMOTE USER LOGIN IN!";
```

Figure 5. String Search Found Other Embedded Login Password in Q-See DVR

5. Experiment

5.1. Overview

We measured the ability of our method by searching the firmware in the real world. We collected six firmware that has a backdoor. Table 2 shows the firmware names and the hardcoded strings. In each firmware, it contains the hardcoded login information.

In this experiment, we ran three methods: whole search, string search, and socket search. In whole search, it searches all functions and finds the function that uses the strcmp or strncmp symbol. String search sets a start point at the strcmp or strncmp symbol and directly searches a function that references the strcmp or strncmp symbol. In socket search, it sets a start point at the socket, searches the function that references the socket symbol, and uses the strcmp or strncmp symbol.

During the experiment, we measured the following values.

- Analyzation time
- Number of functions the search encountered
- Number of functions the search picked as a backdoor candidate
- Number of hardcoded login information the search collected

5.2. Implementation

We developed our tools using the Ghidra plugin (version 9.1). Also, we wrote the plugin with Java SE Development Kit 11 (JDK 11). In our implementation, we used a Windows PC running Windows 10 64-bit, Intel Core i7-7700 3.60-GHz, and 8GB of RAM.

5.3. Measure of Accuracy

To measure our method, we used accuracy, recall, precision, and F-score. Table 3 lists the factors used to calculate the goodness of fit and recurrence rates.

True positive (TP) indicates the number of a backdoor candidate detected by this method, that is, a backdoor.

False positive (FP) is the number of a backdoor candidate detected by this method that is not a backdoor. False negative (FN) is the number of a backdoor candidate that this method does not detect when it is, in fact, a backdoor. True negative (TN) is a type-matched configuration that can be used to indicate the number of pieces not detected by the method.

Accuracy is a percentage of data classified correctly(1).

The relevance of the method is expressed by the precision, which indicates the percentage of type discrepancies detected by the method that is really a backdoor candidate (2).

Recall indicates the comprehensiveness of the method. These values are the percentages of a candidate number under the accuracy-test that we were able to detect with our method (3).

The F-value represents an overall assessment of accuracy and completeness, and the harmonic mean of the fit and reproduction rates (4).

5.4. Whole Search

This method searches all functions in firmware and picks the function and the line that uses the strcmp or strncmp symbol. Table 4 shows the result of the whole search. The Firmware Size column shows the firmware size by KB. The Number of Functions column shows the number that the search encountered. The Retrieval Time column shows how long the search takes time by minute. The Backdoor Detected column shows whether the search found a hardcoded login information. This method found four of the six hardcoded login information. Analysis of time increases if the firmware is more than 4 MB. The largest firmware, which is the Q-see DVR, takes 38 min to complete - this is very time-consuming. A function number is also a huge number, and if we do not know the answer to the login information, it will be very difficult to find.

However, this method did not detect the two firmware, and this firmware did not show the hardcoded login information. We manually checked the hardcoded login information of this firmware, but there were no strings found in the decompiled code.

5.5. String Search

This method starts by searching from the strcmp and strncmp symbols and then picking the function that references the strcmp and strncmp symbols. Two symbols were used to compare the user input and the hardcoded login information. Thus, by targeting the strcmp and strncmp symbols, we reduce the analysis time and maintain accuracy.

Table 5 shows the result. This method found four hardcoded login information, which is the same in the whole searching. The candidate column shows the number of functions that the method thinks as a candidate of backdoor. On the other hand, the analysis time is lesser than that of the whole search. The analysis time of all firmware takes 3–5 s, depending on the size of the firmware.

In this search, five backdoor function from six firmware was founded. The reason why this method was not able to find TCP32764 Router's backdoor is this firmware is a child of backdoor firmware. There is another main firmware that has a trigger to call TCP32764 Router program. Thus, String Search did not find hardcoded login information from the program.

The other result at Ray Sharp DVR, 519070 was reported as an embedded login password according to Table 2. However, the method did not find these strings. We checked manually by using Ghidra to find out the password. Figure 4 shows that the line of 519070 password a string of this line was hidden by value name, so the string method did not find it. We also used IDAPro to compare this problem. IDAPro shows 519070 string in the same line.

In this experiment, we found that String Search is unstable depend on Software Reverse Engineering platform. This search method has room for improvement to convert a string value into a string.

On the other hand, we found other embedded passwords within the same function. Figure 5 shows that 664225 password and ID root was embedded. This result shows that the firmware has several login routes.

Table 4. Result of Whole Search

Firmware Name	Firmware Size (kb)	Number of Function	Retrieval Time (min)	Backdoor Detected
D-Link Router	619	967	2.43	yes
Q-See DVR	7200	7030	38.83	yes
Trendnet Router	318	348	1.45	yes
Tenda Router	566	738	1.83	yes
TCP32764	18	61	0.11	no
Ray Sharp DVR	4900	5535	16.31	yes

Table 5. Result of String Search

Firmware Name	Func	Candidate	Time (sec)	Detected	TP	FP	FN	TN	Accuracy	Precision	Recall	F-score
D-Link Router	205	45	38	yes	1	44	0	762	0.945	0.022	1	0.043
Q-See DVR	833	103	278	yes	1	102	0	134	0.57	0.01	1	0.019
Trendnet Router	168	84	79	yes	1	83	0	799	0.906	0.012	1	0.024
Tenda Router	242	64	37	yes	1	63	0	725	0.92	0.016	1	0.031
TCP32764 Router	11	2	3	no	0	2	1	956	0.997	0	0	N/A
Ray Sharp DVR	866	262	113	yes	0	262	1	101	0.277	0	0	N/A

Table 6. Result of Socket Search

Firmware Name	Func	Candidate	Time (sec)	Detected	TP	FP	FN	TN	Accuracy	Precision	Recall	F-score	Depth
D-Link Router	784	45	79	yes	1	44	0	183	0.807	0.022	1	0.043	4
Q-See DVR	3353	98	402	yes	1	97	0	3677	0.974	0.01	1	0.02	4
Trendnet Router	254	84	50	yes	1	83	0	94	0.534	0.012	1	0.024	4
Tenda Router	31	5	14	yes	1	4	0	707	0.994	0.2	1	0.333	1
TCP32764 Router	29	2	1	no	0	2	1	32	0.914	0	0	N/A	5
RaySharp DVR	3042	262	303	yes	0	262	1	2493	0.905	0	0	N/A	5

5.6. Socket Search

This method starts by searching from the socket symbol and then picking the function that is connected to the socket symbol and using the strcmp or strncmp symbol. It can be noted that backdoors that are reported in previous works have always been accessed via the network. Thus, this method searches the functions that are located around the socket symbol and extract the line that compares hardcoded strings from the user input as the backdoor candidate. In this experiment, we set the depth to five.

Table 6 shows the result. The depth column indicates the depth when the method found hardcoded login information. The candidate number of this method is smaller than that of the string search. This method is effective for Tenda Router, because hardcoded login information was found when the depth is one, which is the minimum candidate. On the other hand, when the depth is four, the number of candidates is high. If there are too many candidates, it will be very difficult to find the login information. Thus, it is necessary to add a parameter to reduce the number of candidates.

At related work, making a model takes hours. In contrast, our model was able to list a candidate of backdoor with a lightweight algorithm, reducing the analysis time. It is necessary to maintenance model to keep an accuracy, but the maintenance is also costly. Also, if learning data are outdated, the model misclassifies it as a backdoor.

Our simple algorithm, on the other hand, helps to analyze the updated firmware of IoT devices without time loss. This system would help the user to quickly analyze the latest firmware.

6. Conclusion

In this paper, we suggested two methods to detect the hardcoded login information-string search and socket search. We focused on the string and network function, which are often used by a backdoor. In string searching, it searches the function of the line that uses the strcmp or strcnmp symbol. As a result, it shows these lines as a candidate of backdoor. In socket searching, it searches

the function that references the socket symbol. Because the backdoors have always been accessed via the network and the network functions use the socket symbol, this method starts from the socket symbol and searches the function of the line that uses the strcmp or strcnmp symbol as a candidate of backdoor.

In the experiment, we used real-world firmware that has hardcoded login information to detect how many backdoors the tool will find. As a result, both the search string and the search socket found hardcoded login information within the search candidate resulting from the five firmware.

In future work, we are going to improve a more precise socket search, and we are going to fine-tune the scope of the search. For example, our method did not find an embedded string when a string is hidden in value. So we are going to improve our engine to convert a string value into a string. There is a big difference between depth one to four, and we need to shorten this difference.

7. Acknowledgments

The authors are grateful to Professor. Sam L. Thomas for suggesting the topic treated in this paper. This work was supported by JSPS KAKENHI Grant Numbers JP17H04705, JP18H03229, JP18H03340, JP18K19835, JP19K12107, JP19H04113, and JST, PRESTO Grant Number JPMJPR1934. 8.

References

[1] Y. Mezquita, R. Casado, A. Gonzalez-Briones, J. Prieto and J. Manuel Corchado. (2019). Blockchain technology in iot systems: Review of the challenges. In Annals of Emerging Technologies in Computing (AETiC), vol. 3, pp. 17-24.

[2] M. Onik, N. Al-Zaben, H. Hoo and C. Kim. (2018). A novel approach for network attack classification based on sequential questions. In Annals of Emerging Technologies in Computing (AETiC), vol. 2, pp. 1-14.

[3] S. L Thomas and A. Francillon. (2018). Backdoors: Definition, deniability and detection. 21st International Symposium on Research in Attacks, Intrusions and Defenses (RAID), pp.92-113, Greece.

[4] S. L. Thomas, T. Chothia and F. D. Garcia. (2017). Stringer: Measuring the importance of static data comparisons to detect backdoors and undocumented functionality. European Symposium on Research in Computer Security (ESORICS), pp. 513–531, Oslo, Norway.

[5] Y. Zhang and V. Paxson. (2000). Detecting backdoors. 9th USENIX Security Symposium (USENIX), vol. 9, pp.12, Denver, Colorado, USA.

[6] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel and G. Vigna. (2015). Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. Network and Distributed System Security Symposium (NDSS), San Diego, California.

[7] J. C. King.(1976). Symbolic execution and program testing. Communications of the ACM, vol. 19, no. 7, pp. 385–394, 1976.

[8] S. Thomas, F. Garcia and T. Chothia. (2017). Humidify: A tool for hidden functionality detection in firmware. Network and Distributed System Security Symposium (NDSS), pp. 279–300, San Diego, California.

[9] J. Salwan, S. Bardin and M. Potet. (2018). Symbolic deobfuscation: From virtualized code back to the original. Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), pp. 372–392, Saclay, France.

[10] M. Yoda, S. Sakuraba, Y. Sei, Y. Tahara and A. Ohsuga. (2020). Detection of the hardcoded login information from socket symbols. International Conference on Computing, Electronics Communications Engineering (iCCECE), pp. 33–38, UK.

[11] J. Ming, D. Wu, J. Wang, G. Xiao and P. Liu. (2016). Straighttaint: Decoupled offline symbolic taint analysis. IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 308–319, Singapore, Singapore.

[12] K. Yakdan, S. Eschweiler and E. Gerhards-Padilla. (2013). Recompile: A decompilation framework for static analysis of binaries. International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE), pp. 95–102, Fajardo, Puerto Rico, USA.

[13] Y. David, U. Alon and E. Yahav. (2020). Neural reverse engineering of stripped binaries using augmented control flow graphs. In Proceedings of the ACM on Programming Languages, vol. 4, pp. 1 – 28.

[14] B. Garmany, M. Stoffel, R. Gawlik and T. Holz. (2019). Static detection of uninitialized stack variables in binary code (ESORICS), pp. 68–87, Luxembourg.

[15] T.Stoenescu, A. Stefanescu, S. Predut and F. Ipate. (2016). River: A binary analysis framework using symbolic execution and reversible x86 instructions. Formal Methods (FM), pp. 779–785, imassol, Cyprus.
[16] Silvio and Cesare. (2013). Bugalyze.com-detecting bugs using decompilation and data flow analysis. Black Hat USA, 2013.
[17] S. Alrabaee, L. Wang and M. Debbabi. (2016). Bingold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (sfgs). DFRWS USA 2016 Annual Conference, vol. 18, pp. S11-S22, Seattle, WA.

© 2021 by the author(s). Published by Annals of Emerging Technologies in Computing (AETiC), under the terms and conditions of the Creative Commons Attribution (CC BY) license which can be accessed at http://creativecommons.org/licenses/by/4.0.