Syed Qasim
Zachary Kimpfler

Detecting Information Leaks in Applications Given Inputs and A Result

Introduction

The Internet of Things (IoT) is the network of devices that are able to connect, collect, and exchange data. This idea has become a buzz word within the Computer Science and Security industry. A large reason why the Internet of Things has gained so much attention is because of the possible security risks that these communicating devices can pose. Many devices have access points or back doors for use by the developers, and these are often unsecured on product release and can be used for malicious purposes. If a node within a network is compromised, the whole network could be monitored and sensitive information could be inferred given the malicious user knows the some of the inputs and outputs of a program. Our project aims to determine, if a given IoT program that contains sensitive information is leaking, is it possible to infer some sensitive information from just observing the input and output of the program with some prior knowledge of what it is calculating.

Related Work

Given the field of suggestion mining is in what could be considered its infancy, there has been only a sparse amount of research done in regards to this topic. However, an article written by Anjaria et al. does attempt to create a method to quantify the amount if information leaked from a given program. Previous studies only focused on determining the information leakage of a certain portion of code within a program, however, the article by Anjaria et al. looks at how that information leak can continue to have effect throughout the program. Anjaria proposes the use of a hierarchical calling tree graph. The tree begins with the root function, or the function that began program execution, and as that function calls others, a tree is formed. The functions are the nodes and the edges are the functions calling one another. This model assumes that the exchange of information between software components is lossless, regardless of transmission medium. Once a call tree graph has been generated, the risk factor of a given node can be determined. If a tree has n levels, and there is a leaking entity at level k where $1 < k < n$, the risk factor is the sum of the risk per call at level i from $i = 1$ to k plus the sum if risk per call at level $j * P$ from $j = k + 1$ to n. P is the risk propagation probability which is a random number decided by the user. The risk per call is calculated by dividing the risk factor at a given level divided by the total number of nodes in the call tree at that level. The risk factor is also a number chosen by the user. These values combined help to summarize the amount of information leaked from a single leaky node, and how that risk increased when the leaky entity communicated with other entities. Chothia et al. made a tool made LeakWatch. It works on programs written in java language and analyzes the quantitative information leakage. LeakWatch is based on a “point-to-point” information leakage model in which secret and publicly-observable data may occur at any time during the program’s execution, including inside complex code structures such as branches and nested loops. The data variables are tagged with ‘secret’ and ‘observed’ given on the sensitivity of the data. Then LeakWatch repeatedly executes it, recording the occurrences of secret and public data, and then performs robust statistical tests to detect whether an information leak is present and, if so, estimate the size of the leak.

Problem Description/Solution

The problem we are attempting to solve is give a piece of software, some inputs, knowledge about the software's internal workings (preferably source code), and output, is it possible to determine the value of one of the input variables that was listed as sensitive is not accessible.Being able to find the value of a sensitive variable with only the output, some inputs, and an understanding of the code base could lead to serious security risks. IoT devices, as they rise in popularity, have continued to not have any standardized protocol for securing communications Developers of IoT products often even go as far as to include back doors in their applications that are also occasionally left completely unsecured. Because of this plethora of devices in which the developer had flagrant disregard for their devices use and security of communication, these back doors serve as portals into networks in which other devices on the network, or communicating with the compromised device, may also be compromised. For reference, in 2016 it was estimated there was over 400 million IoT devices with cellular communication capabilities. Since then, the number has continued its exponential increase. In addition to these devices with cellular communication capabilities, cellphones are also considered IoT devices. In 2018, it was predicted that the number of cellphones would also increase over the 400 million mark world wide. Cellphones are multipurpose devices and therefore have much greater communication and processing capabilities over some IoT devices. However, often these phones have fewer backdoors or flaws in their security features and communications. Despite this, every device is exploitable, given it has a vulnerability, when connected to a network and therefore this large number of cellphones and IoT devices continue to pose a large issue for governments and companies alike who are trying to combat the ever changing landscape that is IoT technology and security risks. These vulnerabilities and growing risk of IoT devices being compromised provided the motivation for this research. Many businesses and governments would be interested in knowing if their applications that are run on their devices have the capability to leak sensitive information. The static analysis performed by our application provides a clear, binary approach to seeing the influence of a sensitive variable throughout an application. The term binary in this sense is meant to imply that, regardless of any condition or the frequency in which a block of code may actually be executed is unimportant. Any leak at all is a risk, in some cases, and therefore a binary representation that does not consider how leaky a section of code is but instead just marks it as "vulnerable" has the clear advantage of not leading developers astray. Any leak can be a vulnerability and therefore, in some cases given the importance of the sensitive information, any leak could be catastrophic. Given the potential security risks associated with a leaky application on an unsecured IoT device, our goal was to produce an application capable of determining the spread of sensitive information within an application. Following this will be a discussion of the implementation and evaluation of our method of detecting the spread of sensitive information within an application.

Evaluation/Implementation

Given our goal of creating a program capable of determining the spread of a sensitive variable within a piece of software, we required some means of parsing the code in order to determine what variables were used within the software, and also how they relate to one another. In short, we needed to parse an applications source code for assignments. Within this category of parsing software, we chose to perform static analysis of the code for several reasons. Static analysis of code is done on the file as it sits in its dormant, non-executed state. When analyzing the code, we do not take into consideration any conditional statements that would restrict when a certain block of code is executed. This can be

advantageous in some cases. Static analysis of code bases requires far less complex parsing capabilities than dynamic analysis. Given that, it is much more realistic to create a static parser. Although it may not determine the likelihood of a block of code executing, it shows all spread of sensitive information clearly. The comparison is clear, binary, either the sensitive variable has mixed with a variable or it has not, regardless of probability of reaching that line of code. In comparison, dynamic analysis reads the code and treats it as if the program were executing given some set of inputs and takes into consideration things like conditional statements that may block off sections of code. Creating software that would be capable of doing that would nearly be equivalent to creating a rudimentary compiler, which is very complex and time consuming. In addition, dynamic analysis, although more representative of the true cases of a programs execution, may not be that advantageous in some situations. If a block of code contains a line that creates a relation between a new variable and the tainted variable, but that block of code is only reached with a probability of 1 in 1 million, the block of code may almost never be reached in a real-world production environment. This small case could mean the dynamic analysis could make this leak less to urgent to fix. That is acceptable, however, it should not be wholly disregarded. In summary, we chose to perform static analysis for we felt its binary representation of the spread of sensitive information was sufficient for our research. Following this, we chose a language to develop in and a language to parse. One of the first languages we considered was Java. Java is a popular, multi-platform, easy to learn programming language with very clear rules on how to implement and define objects, constructors, variables, etc. Other languages considered were deemed too flexible or had unwanted features for the purpose of this research. For instance, Python's data types are inferred. It would be difficult to detect the definition of new variables. Out system looks for key-words along with operators. We also considered C or other C variants, but they are extremely flexible in the way you are able to write your code, therefore, we deemed its variability too troublesome to handle. We felt if we pursued C, it would require character by character parsing which would be very time consuming and bordering on creating a compiler to achieve this. Overall, Java fit well for this application as both the language for our parser and for our parsers input language due to both its simplicity and familiarity to the researchers. Before continuing on, there was one major assumption that we defined at this point. Despite Java having well defined rules for creating new variables, objects, classes, etc, there was still a large amount of flexibility that we would be unable to account for without creating a parser that reads the source code in a near dynamic fashion. One of the largest problems stemmed from the use of methods/functions within applications. Parameters of a function are often not the same name as the variable passed to them. Given this lack of consistency between variables passed to functions and the names of the variables within the function, it adds a layer of complexity to following those interactions. To avoid the complexity of this situation given the time restrictions of our research, we opted to ignore functions interior contents and instead assume any function that is passed a tainted variable can only produce tainted output, regardless of the actual code within. After defining this rule, we began our development by planning our method in which we would explore a programs source code. We begin at the class level. The user lists the files they wish to parse, our parser opens the files one by one, reading in each line and analyzing the information within. Each line is considered individually and searched for an equality operator ("=", "+=", "-=", etc...). However, lines that are individually commented or have a block comment on them are ignored. At this stage in the parsers execution, only lines containing an equality remain in the applications memory. All other lines are disregarded. The application then creates Variable objects to hold information relevant to the assignment. Relevant information includes the name of the class that the variable is contained within,

the name of the variable, its type, and its "value" or the right hand side of the equality. We then iterate over this list of Variables, to extract the relationships held within their values, We inspect the right hand side of the equality by parsing it for existing variable names, and in addition to that check for function calls, or the use of objects. These extracted relations are then listed within each Variable object. This information alone is enough to run a final function to determine which variables interact with a sensitive or "tainted"variable. This is then passed to our graphing function for plotting the relations between all variables and marking the edges in which a sensitive variable interacts with a previously nonsensitive variable. In addition, one last minute feature added to the application allows it to create a Gephi ".gexf" file. Gephi is a free graph visualization software available at the Gephi website (https://gephi.org/). Gephi offers much more flexibility in terms of modeling your graph. However, there are a few limitations of using this feature of our application. The output Gephi ".gexf" file does not contain taint spread information, given restrictions on functionality due to time constraints. Despite this small limitation, the underlying application still provides the necessary information to determine which nodes are tainted via the console output. In addition, Gephi's increased graphing capabilities makes up for the rudimentary graphing offered by our applications built-in graphing software. Finally, our algorithm does some minor analysis of each file it has parsed. Information calculated ranges from total number of nodes, tainted nodes, in-degree and out-degree of a node, along with the average across the whole class. The total in-degree and out-degree of each of the class represents the total edges pointing into and out of a node. In addition, average number of tainted nodes, and average in and out degree is listed as well. This information helps the user to get a better understanding of exactly how great an influence the sensitive variable has on the code base as well as how connected their variables are.

Discussion/Conclusion

Following completing the implementation of the parser, analysis on several more sample applications was done to test how well it performs on previously unseen code. In addition to determining how well our parser handled unique code styles it had not seen before, we also looked at the analysis performed by our application. Table 1, below, displays the output of the analysis of five sample programs. From the results shown, there are some notable differences. The file created explicitly for testing this application during development shows 26% of its variables have, at some point, interacted with a sensitive variable and have been marked tainted. In comparison, all other analyzed programs were written normally, and show a much smaller level of interaction among sensitive and non-sensitive variables. Possibly one of the most interesting sets of results is those for "Parser.txt". This file is the source code of this project's parser. It boasts the largest number of variables, and highest in-degree and out-degree of all programs not explicitly written for development of this parser. It is reasonable to believe, these two factors, in conjunction, would make the program to have a large number of tainted nodes. However, this is not the case. "Parser.txt" demonstrates that it is very possible for applications to have very distinct branches of variable interaction in which very little interaction between branches may occur. Also, some nodes that are tainted may have a very low out-degree and therefore have very little influence on the other variables within its branch. The result provided in the *Table 1* indicate that the out-degree of the sensitive variable determines the extent of information leakage. If the number of variables using realing on the sensitive variable are high then there is a higher chance of information leakage.

File Name	# Variables	Sensitive Variable Name	# Tainted (Sensitive) Variables	Avg. In-Degree of Nodes	Avg. Out-Degree of Nodes	Avg. # Tainted Nodes
ExampleProgram.txt	15	C	4	0.733	0.733	0.26
Mastermind.txt	56	computerCode	1	0.267	0.267	0.017
A1_Q2.txt	9	Counter	1	0.11	0.11	0.11
GA.txt	40	newx	1	0.22	0.22	0.025
Parser.txt	50	s	1	0.32	0.32	0.02

Table 1.

In addition to the numerical analysis provided in the console output of our application, the graphs created also offer a visual aspect to aid the user in determining the spread and interaction among variables. *Figure 1* is a graph produced by Gephi with some minor customization. In this graph, a node with a dark red color indicates a high out-degreem, where as nodes with a lighter color have a lower out-degree.

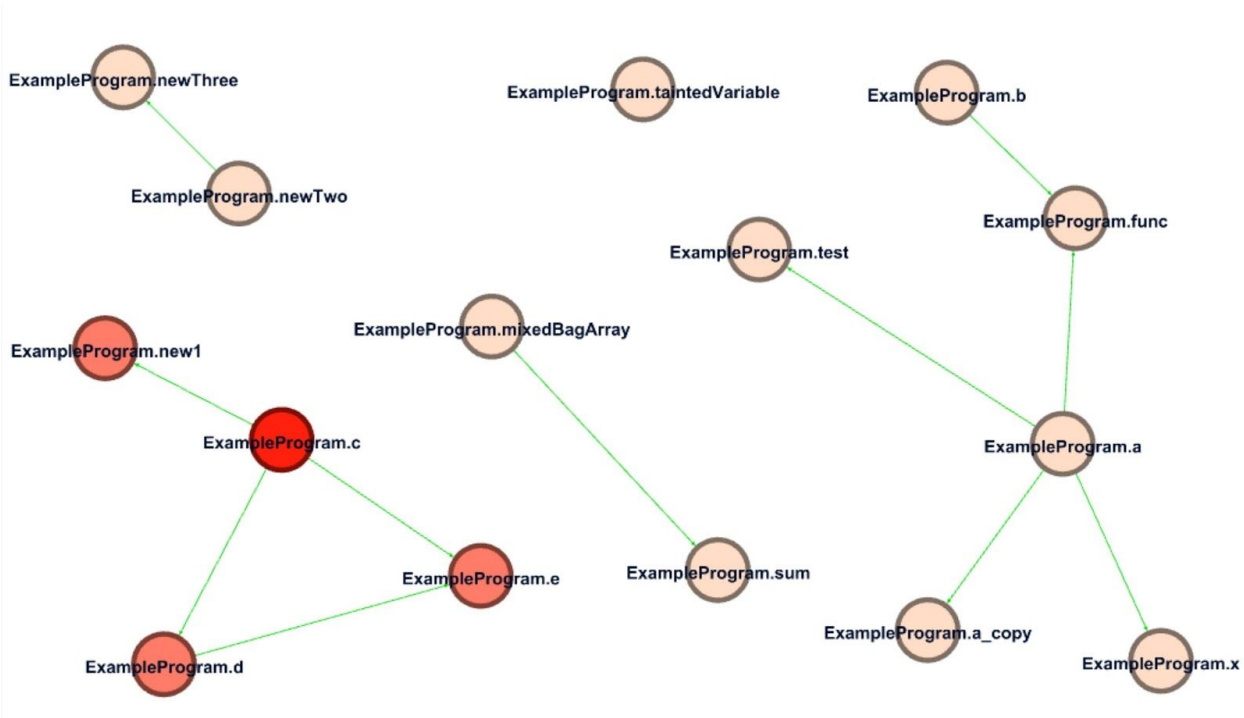

Figure 1.

Figure 1 shows the graph of gephi output generated by our application for the example program. The direction of edges shows the in and out degree of the nodes. Variable a and c are have the higher out-degree then the rest of the nodes. For you analysis we picked variable c as sensitive variable. Since three other variables are connected with variable c, the graph shows the taint spread or the possible sensitive information leakage. In case of an IOT program, if the programmer does not sanitize these tainted variables and they are open to public, an adversary can use the these tainted variables to infer the information about the sensitive data. This parsing application can help the user visualize the source code and identify the possible inference channel with in the source code. Moreover this application can also help in debugging complex or large source code files.

The original task of this research was to determine, given non-sensitive and sensitive input, output, and knowledge about an application's source code, is it possible to determine the value of the sensitive variable indicated. Overall, it is reasonable to state that yes, it is possible to work in reverse to determine the value of a sensitive variable that was previously unknown. That being said, there are many restrictions and factors that can greatly influence how feasible this task may be. For the purpose of this research, the values that were marked as sensitive were often simple data types. In a real-world scenario, data types may be far more complex. Data such as video, audio, or other stream-like or modified data may be the output of an application and would be extremely time consuming to reverse-engineer. If the input and output data consists of simplistic data types, and the code base and variable relations are not overly complex, working backwards to the value of the sensitive variable would be nearly trivial.

References:

1. Anjaria, Kushal, and Arun Mishra. "Information Leakage Analysis of Software: How to Make It Useful to IT Industries?" *Future Computing and Informatics Journal*2, no. 1 (June 2017): 10-18. Accessed September 30. doi:10.1016/j.fcij.2017.04.002.
2. Chothia, Tom, Yusuke Kawamoto, and Chris Novakovic. "LeakWatch: Estimating Information Leakage from Java Programs." *Computer Security - ESORICS 2014 Lecture Notes in Computer Science*, 2014, 219-36. doi:10.1007/978-3-319-11212-1_13.
3. "IoT: Number of Connected Devices Worldwide 2012-2025." Statista. https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/.