

Thesis Report

On

LIGHTWEIGHT CODE SELF VERIFICATION FOR INTERNET OF THINGS(IoT) DEVICES

Submitted in partial fulfillment of the requirement of
BITS F421T THESIS

By

**Prashast Srivastava
2012A7TS087U**

**Under the supervision of
Prof. Chittaranjan Hota
BITS-Hyderabad Campus**



ACADEMIC RESEARCH DIVISION

**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
DUBAI CAMPUS, DUBAI, U.A.E.**

September 2015 – January 2016

BITS, Pilani – Dubai
Dubai International Academic City, Dubai

I- Semester 2015-16

Course Name: First Degree Thesis and Seminar Course

Course No: BITS F421T

Duration: 1st August, 2015 – 25th December, 2016

Date of Start: 1st August, 2015

Date of Submission: 23rd December, 2015

Title of the Report: Lightweight Code Self-verification for Internet of Things (IoT) devices

ID No. / Name of the student: 2012A7TS087U / Prashast Srivastava

Discipline of Student: B.E. (Hons.) Computer Science

Name of the Project Supervisor: Prof. Chittaranjan Hota

Key Words: Return Oriented Programming, Code Checksum, Internet of Things

Project Area: Developing self verifying code for Internet of Things (IoT) devices

Abstract: This research work deals with developing lightweight code self verification methods for Internet of Things(IoT) devices. The Internet of Things is the network of physical objects embedded with electronics, software, sensors to enable these objects to achieve greater value and service. In a hostile environment, adversaries may tamper with the devices for their own interest. The idea is to use Return Oriented Programming as a defensive weapon to protect programs running in hostile environment. Code checksum has been used to protect the critical regions of code as well so as to ensure that there are multiple lines of defense setup to protect the code from being tampered.



Signature of Student

Date: 23-12-2015



Signature of Supervisor

Date: 23-12-2015

Acknowledgements

I would like to express my heartfelt gratitude to Prof. Dr. R.N. Saha, Director BITS Pilani, Dubai Campus who has given us the opportunity to apply the knowledge acquired at college and gain further practical knowledge.

My sincere gratitude to Dr. Neeru Sood, Associate Dean, Academic Research Division and Prof. Chittaranjan Hota, my Supervisor, for providing me with all assistance required to perform my best during the project execution. He has been a great mentor and guide throughout, always ensuring that I am able to overcome all the obstacles along the way. I would also like to thank Dr. Alamelu Mangai for her constant support throughout the project without whose supervision this would not have been possible at all.



Prashast Srivastava
2012A7TS087U

**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
DUBAI CAMPUS, DUBAI, U.A.E.**

CERTIFICATE

This is to certify that the thesis entitled **Lightweight Code Self Verification for Internet of Things (IoT) devices** and submitted by **Prashast Srivastava** ID No. **2012A7TS087U** in partial fulfillment of the requirement for BITS F421T. Thesis embodies the work done by him under my supervision.

Date: 23/12/2015



Signature of the Supervisor

Name- Prof. Chittaranjan Hota

Designation:- Professor

Department of Computer Sc. & Engineering
Associate Dean (Admissions)
BITS-Pilani, Hyderabad campus.

Table of Contents

Title	Page No.
Chapter 1 Introduction	1
Chapter 2 Return Oriented Programming	3
2.1 ROP as Code Verification Technique	5
2.2 ROP Defence Vulnerabilities	8
Chapter 3 Code Checksumming	9
3.1 Cross Verifying Guard Framework	10
3.2 Code Checksumming with ROP	11
3.2.1 Implementation	12
3.3 Code Checksum Vulnerabilities	14
Chapter 4 Algorithm for implementing ROP & Code Checksum	15
Chapter 5 Performance Analysis	18
Chapter 6 Implementation of Security on IoT testbed	20
6.1 The IoT node- Intel Galileo Board	20
6.2 The IoT testbed	21
6.3 Methodology for running security mounted code on IoT	22
6.3.1 Compiling code on the Galileo board	23
6.3.2 Compiling code on the Workstation(PC)	23
6.4 Penetration Testing	24
6.4.1 Inserting code within the protected code region	25
6.4.2 Inserting code outside the protected code region	26
6.4.3 Inserting code in a different linked C program	27
6.4.4 Tampering with the checksum keys	28
Chapter 7 Scope for Future Work	29
Chapter 8 Conclusion	30
References	31

LIST OF FIGURES

1. ***Figure 1*** Sample ROP chain
2. ***Figure 2*** Hello World program with ROP functionality
3. ***Figure 3*** Stack View
4. ***Figure 4*** ROPGadget
5. ***Figure 5*** GDB Debugger
6. ***Figure 6*** Implementation of ROP
7. ***Figure 7*** Code Checksumming
8. ***Figure 8*** Implementation of ROP with Code Checksumming
9. ***Figure 9*** Sample guard template
10. ***Figure 10*** Checksum Code Vulnerability
11. ***Figure 11*** Reworked Stack
12. ***Figure 12*** ROPgadget-gadget finder
13. ***Figure 13*** XOR Tester
14. ***Figure 14*** Intel Galileo Board
15. ***Figure 15*** IoT testbed network structure

1. Introduction

The Internet of Things (IoT) is an interconnected network of boards referred to as nodes which have a variety of modules attached to them in the form of sensors (temperature, heat, energy sensors), internet connectivity(LAN port), Linux and Arduino compatibility as well which enables these nodes to collect data, send it to the servers and accordingly make visualizations based on the data which is collected. This kind of a framework helps in collecting data remotely from distant locations and use it for visualization. The boundaries imposed by a legacy framework vanish which enables us to integrate the physical world around us with these Internet of Things (IoT) nodes which operate throughout the physical space. Each node has it's own unique identity owing to the different MAC addresses assigned to each node as well as a unique digital footprint.

The interconnectivity flaunted by this IoT framework holds a lot of potential and real-time applications in our physical world. It goes beyond the capabilities of a normal Machine to Machine network which merely allowed wireless and wired systems to communicate with each other with a limitation that the devices communicating had to be of the same type. The possible applications of this IoT framework are endless.

They can help us in realizing the concept of a smart city. A city that is able to use the data collected from the IoT nodes, make inferences based on this data and fine tune it's functioning autonomously whether it be urban services or optimizing the resource consumption and energy usage. This framework would not only help in optimizing existing resource usage but also will help in allowing for real time response to challenges. These cities will also encourage participation from the local people and use the collective intelligence to build a better city. Initiatives towards realizing this goal have already been started to be taken with many smart cities being developed in Amsterdam, Barcelona, Stockholm, Santa Cruz to name a few.

But as any technology which holds a lot of potential, it also has it's own set of drawbacks and concerns which need to be addressed before we can move forward and start implementing it on a large scale. One of the main issues which need to be dealt with is privacy. As we know, that such a framework will be collecting huge volumes of data and this data, if it falls into the wrong hands will have catastrophic impacts with the privacy of a millions of consumers compromised. In addition to that such a framework if implemented will have autonomous control over certain physical aspects. Let us take the example of a smart city, there will be certain nodes which will have the ability to regulate the power usage. What if these nodes are compromised and the controls fall into the hands of a malicious entity? These are concerns which need to be addressed straight away and the work that has been done in this project serves to answer some of the concerns which have been raised.

Everyone ranging from application vendors to administrators and users require methods to safeguard their code against malicious users. Research into security measures to protect the software and the critical code has also increased drastically of late mainly because of the high magnitude of data and content in the form of software which is being shared across the Internet as well as the need to harden the security measures for a personal computer, servers etc. And

now with the advent of Internet of Things and its proliferation requires us to amplify our research efforts even more.

Code verification is a subclass of techniques which can be used to ensure up to a certain extent that the code which is being executed on an IoT node is the untampered code which is intended to be run on the node and that it hasn't been tampered with by some malicious entity. Self verification is a subclass of code verification. This technique works around the concept that the code which needs to be protected should be able to verify its own integrity. In order to verify the integrity of the code generally more code has to be added to the code to be protected which leads to unnecessary overhead. This is an overhead which we aim to minimize by using Return Oriented Programming along with code checksumming so as to develop a lightweight code self verification technique which can be used with minimal overhead and won't pressurize an IoT node too much.

2. Return Oriented Programming

Return Oriented Programming is originally an exploit technique to bypass computer security measures such as Data Execution Prevention($W\oplus X$) and Address Space Layout Randomization(ASLR) and allow for malicious code execution. This technique uses the buffer overflow vulnerability to gain control of the call stack and then execute carefully chosen short return terminated machine instruction sequences called “gadgets”[1].

In order to mitigate such exploits originally it was thought that Data Execution Prevention protection was enough under which memory is either marked as writable or executable but may not be both. Thus, a hostile party can not come in simply, place it's payload and execute it by manipulating the instruction pointer to it. In addition to that another security measure in the form of Address Space Layout Randomization was placed as well in order to lessen the chances of mounting such an attack. If this is implemented, the virtual address space which is allotted to the program every time it's executed is kept on being randomized which in turn leads to the gadget addresses being randomized and as such it becomes increasingly difficult to mount a ROP attack to execute a malicious payload.

However through numerous studies conducted in this regard it has been proven that it is indeed possible to mount an ROP payload attack even in the presence of such security measures. How do they make it possible? In order to circumvent the Data Execution Prevention protection what the hostile parties have started doing is using this technique known as the Ret2LibC attack. Since an attacker can't place his own payload on the stack and execute it owing to the $W\oplus X$ protection, what he does is use the memory segments which have already been marked executable and use the gadgets in these memory locations to mount his attack. He uses the gadgets present in the program code itself along with the shared libraries which often had subroutines to perform functionality potentially useful to a malicious user. Instead of writing his own payload onto the stack and overwriting the Extended Instruction Pointer(EIP) with it's address what the hacker does is he overwrites the EIP with the the address of gadgets available in the already present library functions.

As far as Address Space Layout randomization is concerned, even though it does make it harder for an attacker to mount a ROP payload since the address of the gadgets keep on getting randomized again and again with each execution as the shared libraries keep it has been found that it is possible to mount an attack even with this security measure on. How? ASLR is vulnerable to information leakage attacks and other approaches to determine the address of any known library function in memory. Therefore, if an attacker is able to successfully determine even the address of one instruction, he will be able to infer the address of other instructions as well and will be able to mount a ROP payload attack. In order to thwart these kind of advances, another thing which could be done is to relocate all the program instructions separately but then again that places a lot of overhead on the processor and can lead to severe runtime inefficiency. Moreover, it has been found out that even OS such as Windows and Linux which implement this ASLR, while doing so don't randomize the entire program space for the reason listed before and as such the attacker even without inferring addresses can just use the code chunks in the unrandomized code to construct his attack.

The organizational structure which is used to construct the attack is known as a gadget which are these short return terminated instruction sequences. It is necessary for them to be return terminated so as to ensure that function calls can be chained. It has been shown that these gadgets provide Turing complete functionality which in turn means that it is possible to execute logical constructs such as conditional branching, iterative loops using these gadgets alone and that too even when security protections such as $W\oplus X$ and ASLR are working.

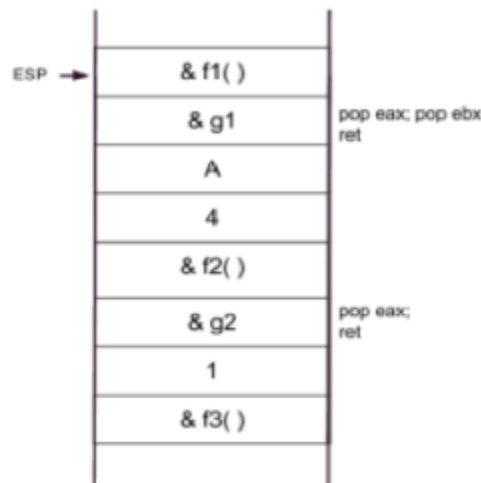


Figure 1

Figure 1 illustrates an example of a ROP chain and how gadgets can be used to chain function calls. Suppose we have overflowed the buffer and have been able to overwrite the instruction pointer with the address of f1(). Now how do we get the instruction pointer to jump to f2() and f3() once f1() has been executed? We definitely can't overwrite the buffer again, therefore the answer is gadgets which have been used to jump from f1() to f2() and then to f3().

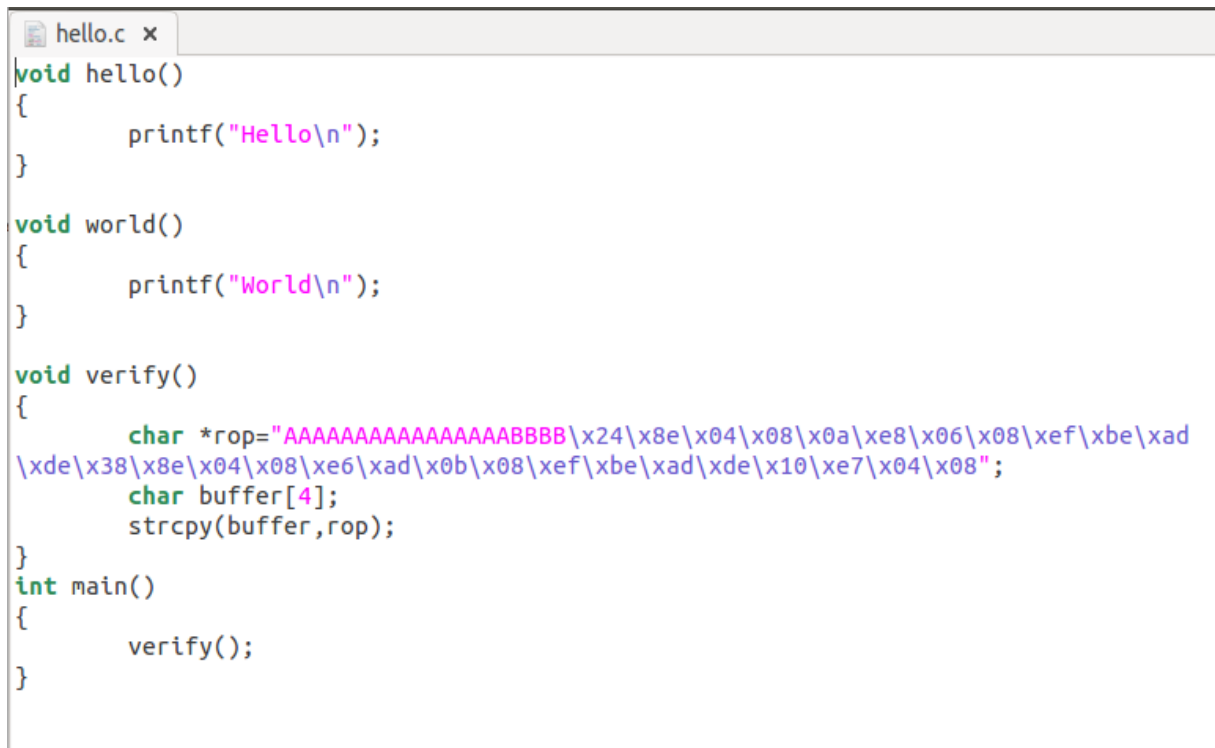
What is happening is after f1() is executed, the control goes to the gadget which in this case is the “pop-pop-ret” gadget. Once that is done, dummy values which have been fed by us into the rop chain get popped into eax and ebx respectively and when the ret instruction comes, the esp is pointing to the starting address of f2() which again has been fed by us into the rop chain and consequently into the call stack. Similarly after f2() has been executed we use another “pop-ret” gadget to transfer the control flow to f3() and henceforth our desired control flow is achieved.

2.1 ROP as a Code Verification Technique

Now initially being used as an exploitation technique, researchers have found out that it can indeed be used as a code verification technique as well by passively verifying the integrity of the protected code.

How do we do that? We protect code by overlapping ROP gadgets with it. Then, selected instructions from the protected program are translated into ROP chains which use the overlapping gadgets. Since tampering with the gadgets causes the translated instructions to malfunction, this implicitly verifies the integrity of the protected code. Thus, we refer to these translated instructions as verification code.[2]

We present this simple hello world program which has been modified with ROP functionality and as such if anyone tries to tamper with the code, a segmentation fault occurs and we get to know that someone has tampered with the code.



```
hello.c x
void hello()
{
    printf("Hello\n");
}

void world()
{
    printf("World\n");
}

void verify()
{
    char *rop="AAAAAAAAAAAAABBBB\x24\x8e\x04\x08\x0a\xe8\x06\x08\xef\xbe\xad
\xde\x38\x8e\x04\x08\xe6\xad\x0b\x08\xef\xbe\xad\xde\x10\xe7\x04\x08";
    char buffer[4];
    strcpy(buffer,rop);
}

int main()
{
    verify();
}
```

Figure 2

Figure 2 shows the hello world program which has been infused with ROP functionality. The simple task of printing “hello world” has been protected by a ROP chain and as such if anyone tries to tamper with the “hello” or the “world” module, segmentation fault occurs, an error is raised and we know that someone has tampered with the program.

Let us go through in detail as to how ROP functionality is being used to implement a simple hello world program. As you can see in Figure 2, through the main function the “hello” and the “world” module aren't being called but instead a verify function is being called which contains the ROP chain and which carries forward the task of printing hello world.

Once the control reaches the verify function, the ROP chain that we've constructed is copied into the buffer. Since the ROP chain size exceeds the buffer size, adjacent memory locations are overwritten and we are able to manipulate the instruction pointer according to our liking. The stack looks like Figure 3 once the ROP chain has been copied onto the buffer array.

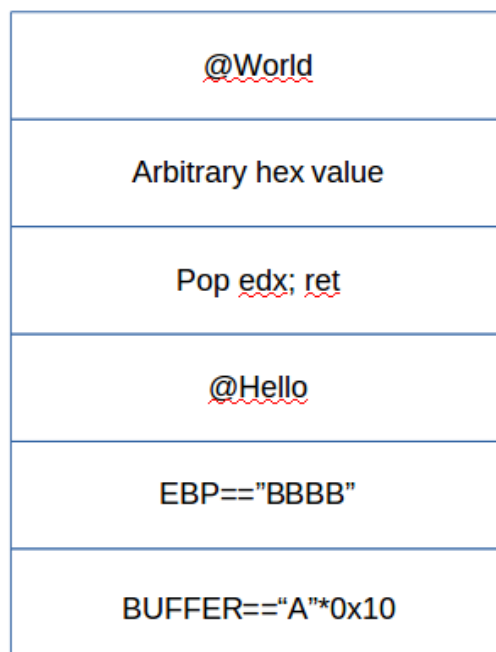


Figure 3

As we all know when a function is called the following things are pushed onto the stack in the following order:-

- Parameters of the function
- Return address(EIP)
- EBP
- Local variables

We have to keep in mind as well that the stack grows in a downward fashion as well. So once we have the structure in mind, we construct the ROP chain accordingly in such a way such that

- Buffer is filled up to it's capacity("A"*0x10)
- EBP is filled with "BBBB" since it's 4 bytes long
- EIP is overwritten with the address of the "Hello" module
- A "pop-ret" gadget is used to chain the "World" module function call
- A junk value is pushed to the stack which will be popped
- Address of "World" module is pushed to the stack which the gadget will return to

Now in order to find out the address of the gadgets in the executable, we use an automatic gadget finder known as ROPGadget as shown in Figure 4 . There are other open source utilities as well but we have used this for the purpose of this experiment.

```
[+] Gadget found: 0x809a13d mov dword ptr [edx], eax ; ret
[+] Gadget found: 0x806e80a pop edx ; ret
[+] Gadget found: 0x80bade6 pop eax ; ret
[+] Gadget found: 0x8054230 xor eax, eax ; ret
```

Figure 4

And in order to find out the addresses of the modules which will need to be in the ROP chain were found out using the GDB debugger as shown in Figure 5.

```
prashast@prashast-X550LD:~$ gdb -q hellorop
Reading symbols from hellorop...done.
(gdb) print hello
$1 = {void ()} 0x8048e24 <hello>
(gdb) print world
$2 = {void ()} 0x8048e38 <world>
(gdb) quit
```

Figure 5

Now as test cases, we put code into the hello module or the world module and tried executing the code, a segmentation fault was raised and we were able to find out that someone had tampered with the code as shown in figure 6.

UNTAMPERED CODE

```
prashast@prashast-X550LD:~$ ./hellorop
Hello
World
```

TAMPERED CODE

```
prashast@prashast-X550LD:~$ ./hellorop
Hello
Segmentation fault (core dumped)
```

Figure 6

2.2 ROP defense vulnerabilities

Even though Return Oriented Programming provides a lightweight method of passively identifying when a program has been tampered there are certain shortcomings which riddle this security technique.

- 1) There is a possibility that the attacker identifies the verification code where we have used ROP functionality and completely circumvent it by displacing it
- 2) The attacker modifies the program in such a way that the modifications reside completely outside the areas which are protected by our ROP functionality
- 3) The attacker modifies the protected code in such a way that the resulting gadgets do not affect the outcome of the verification code
- 4) The attacker makes modifications to the protected code such that the resulting gadgets are semantically equivalent to the original ones

Despite these shortcomings, these conditions severely restrict the modifications which can be made to the program making it that much harder for a malicious entity to tamper with our program.

3. Code Checksumming

As we all know during compilation, the code written in a High Level Language(HLL) firstly gets converted into assembly language from which it is finally converted into binary which is interpreted by the processor. Every code line in assembly language is interpreted as a string of opcodes which in turn let the processor know what exactly is need to be done.

This is the underlying principle that the code checksumming works on. By this technique each code line to be protected interpreted as a string of opcodes is added up which in turn in the end gives a checksum value in hexadecimal format. Verify that the checksum value calculated is the same as that of untampered code section and you verify the integrity of the code which is protected.

Since this is a tamper-resistance technique, it should be able to verify the integrity of the code which it is protecting(code signature) and in addition to that take appropriate actions when tampering has been found. The whole mechanism can be divided into a detection part and an action part. The detection part should be able to recognize modifications on the code and trigger the action part [3]. The detection part can be further divided into two steps, firstly, we need to obtain a signature of the critical code which is needed to be protected. Secondly, verify the integrity of the signature to ensure that the critical piece of code hasn't been tampered with in anyway. In our case, the detection is done using these code segments known as “guards” which checksum the piece of code which is needed to be protected and carry out the appropriate actions if they find out that the code has been tampered with as you can see in Figure 7.

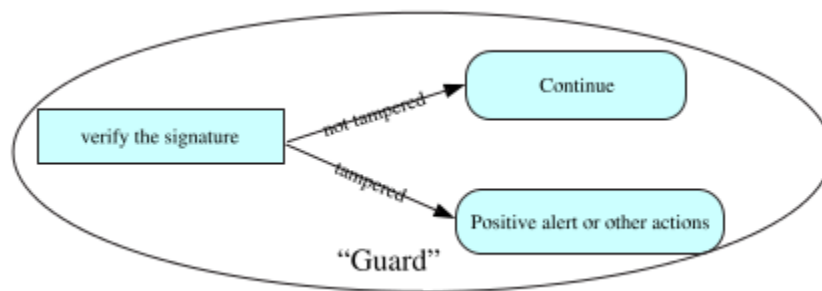


Figure 7

3.1 Cross Verifying Guard Framework

Software cracking is a problem which is prevalent today. It is an exploit in which the malicious entity gets a copy of the software and is successful in breaking the protection mechanism which had been implemented in the software leading to the entity being able to make modified copies of the software.

In order to ensure that doing this kind of a exploit becomes inherently hard and time-consuming for the attacker as a result of which deterring him from carrying through with the exploit, Hoi Chang and Mikhail J. Atallah proposed a network of cross verifying checksum guards which raises the bars for the attackers and makes it “harder” for an exploiter to exploit the code.

In this network, the guards responsible for checksumming the code protect each other as a part of a strongly connected network. This network is harder to bypass because the security is distributed among all the guards. Moreover, there are many permutations and combinations which can be applied while making a guard network which makes it even harder for a malicious entity to penetrate it. And the level of security can always be customized by the user by simply adjusting the number of guards that will form a network of security.

Before going any further, let us just define the role of the guards clearly. A guard is a piece of code responsible for checksumming a piece of code and performing predefined actions if tampering is observed. In our case, the appropriate action which has been predefined for the guard to take is to corrupt the stack pointer so that the program can not progress any further. However the severity of actions can be varied from the mildest of just making a log of the event to the extreme of corrupting the stack pointer. If no tampering is observed then the program execution proceeds as intended.

A group of guards can provide a more resilient form of protection as opposed to a single guard. Some of it's advantages have been listed down below:-

- If a program has multiple regions of code which need to be protected then multiple checksumming guards can be deployed to protect the critical sections of code
- There is no single point of failure for a guard network since the guards are invoked at different points in the runtime so it is not that easy to detach the critical code from these checksumming guards
- There a myriad ways in which a strongly connected guard network can be configured. Different kind of formations can be formed from the guards at our disposal which will make it harder for a hacker to mount an attack
- Based on the level of protection which is required the number of guards can be increased accordingly

3.2 Code Checksumming with ROP

Though return oriented programming in itself is an efficient way to passively identify when someone has tampered with your program but still as a standalone technique it has certain vulnerabilities which we have discussed earlier in this paper. One of the vulnerabilities which had been pointed out was that the attacker might identify the verification code where our ROP functionality had been implemented and circumvent it completely or reverse engineer our ROP code and mimic whatever was being done by our ROP code in a High Level Language.

If the attacker is successful in doing either of these things then the malicious entity can go forward and tamper with the critical piece of code that the ROP chain was protecting. In order to avert these kind of situations it would be useful to deploy code checksumming in the form of cross verifying guard network over the critical pieces of code in conjunction with the ROP functionality.

There are certain advantages which are observed if we implement these two techniques in conjunction with each other:-

- There is no single point of failure. If the attacker is able to circumvent the ROP functionality then he still has to bypass the code checksumming guard network before he can tamper with the critical code. Similarly, just bypassing the guard network and tampering with the critical code won't be of any help because our ROP functionality will identify that the code has been tampered with and perform the appropriate actions.
- The only way for an attacker to bypass this is to simultaneously disable the guard network as well as the ROP functionality which will be time-consuming.
- The protection provided by the cross verifying guard network extends even outside the code region to be protected because the addresses of the flags get changed inside the protected region of code if someone tampers with the code outside of the protected region which in turn is detected by our guard network because the checksum calculated changes
- Putting these cross verifying checksum guards in the assembly level code instead of putting into the source code is useful because it isn't that simple to decipher then where the guards are and where the ROP functionality has been implemented. Therefore, it brings a layer of code obfuscation to the table as well

3.2.1 Implementation

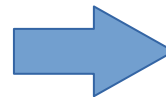
In order to implement ROP functionality with code checksum there are certain procedures which need to be followed. The necessary steps have been listed below in Figure 8.

```
void print_secure()
{
    int a=2;
    a=(a*2)+5;
    int t;
    for(i=0;i<5;i++)
    {
        printf("Data\n");
    }
}

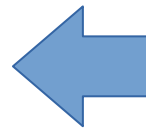
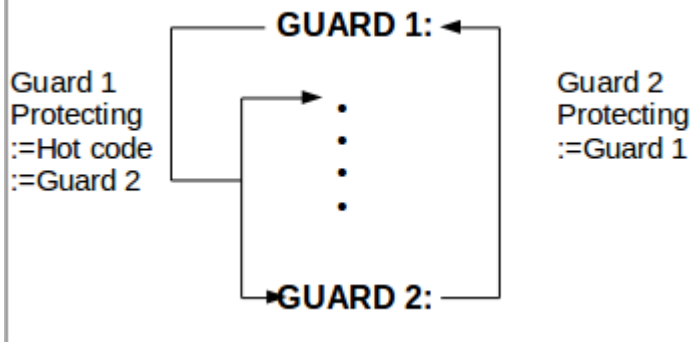
void verify()
{
    char *rop="AAAAAAAAAAAAABBBB\x1a\xe8\x06\x08\x60\x0e\x08\xf6\xad\x0b
\x08Yes!\x4d\xa1\x09\x08\x30\xf6\x04\x08\x1a\xe8\x06\x08\x60\x0e\x08\x24\x8e\x04\x0b
\xf6\xad\x0b\x08\xef\xbe\xad\xde\x20\xe7\x04\x08";
    char buffer[4];
    strcpy(buffer,rop);
}

int main()
{
    verify();
}
```

Create source code with ROP functionality



Generate assembly level code of the program



Put cross verifying checksum guards around the hot pieces of code(for eg. The for loop) in the program

Figure 8

Let us elaborate further on the steps which need to be followed in order to implement ROP with code checksum on a program for enhanced security.

- Firstly select a verification function from the given code and implement that functionality using ROP and using the gadgets from the critical pieces of code which need to be protected
- Generate an assembly level code for the modified source code infused with ROP
- Implement a cross verifying checksum guard network protecting the critical piece of code.
- You will need to make appropriate modifications to the ROP chain that you've used since the addresses of the gadgets will change
- Place XOR-encrypted values of the correct checksum values(key) with pre-decided values on the assembly level code as well so as to ensure no one tampers with the keys and take appropriate actions if tampering with the key is evident

For experimental purposes we had implemented a simple 2-guard strongly connected network which was working in conjunction with the ROP functionality that we had implemented. The sample guard template which we have used in our program is shown in figure 9.

```
guard:
    movl .checksum,%ebx    #initialising ebx(checksum reg) to 0
    movl $client_start,%ecx
for:
    cmpl $client_end,%ecx  #if ecx greater than client_end
    jg end
    addl (%ecx),%ebx
    addl $4,%ecx
    jmp for
end:
    cmpl .key,%ebx #comparing calculated checksum with existing value
    je succ
    addl %ebx,%ebp #corrupting ebp if tampering found
succ:
    jmp .L2
```

Figure 9

3.3 Code Checksum Vulnerabilities

Even though Code Checksumming works relatively well against static tampering of the code, it does not protect against dynamic tampering.

Wurster et.al were able to find a vulnerability in this security procedure. All the checksumming procedures that have been developed up until now work under the assumption that the code which is read for the purpose of checksumming is same as the code which is executed by the processor.

They were able to find out that this assumption doesn't hold true for most of the modern processors as their design makes this assumption flawed[4]. Therefore, if an attacker holds administrative privileges, he can direct the kernel to ensure that when the guards are calculating the code checksum they read the untampered pieces of code but when the instructions are being fetched, the processor should take them from the malicious piece of code, hence ensuring that the code checksum procedures don't raise an alert.

The methodology depicted above has been shown diagrammatically in the figure below.

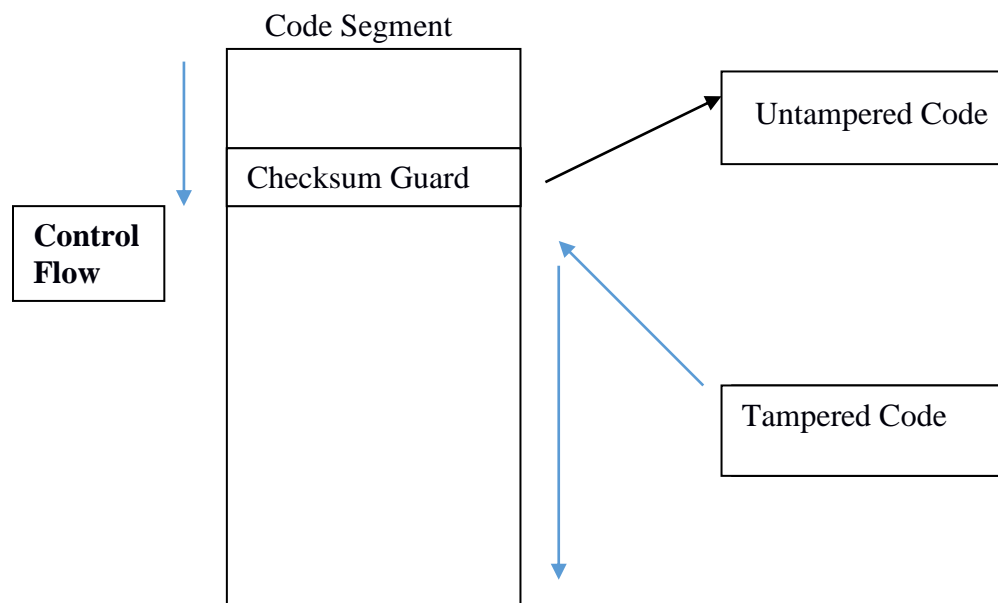


Figure 10

4. Algorithm for implementing ROP and Code Checksum

There are certain steps which need to be followed if we are to implement our dual security measure of ROP functionality along with code checksumming onto a C program.

1) Modify the given C program to put a ROP template in it. When I say a ROP template, what I mean is a function that is implementing ROP functionality to do some arbitrary instructions (in this case print 'yes') and then jump to *print_secure*, the original functionality of the program.

```
void print_secure()
{
    int a=2;
    a=(a*2)+5;
    int i;
    for(i=0;i<5;i++)
    {
        printf("Data\n");
    }
}

int main()
{
    print_secure();
}
```



```
void print_secure()
{
    int a=2;
    a=(a*2)+5;
    int i;
    for(i=0;i<5;i++)
    {
        printf("Data\n");
    }
}

void verify()
{
    char *rop="AAAAAAAAAAAAABBBB\x1a\xe8\x06\x08\x60\xa0\x0e
\x08\xf6\xad\x0b\x08Yes!\x4d\xa1\x09\x08\x30\xf6\x04\x08\x1a\xe8\x06
\x08\x60\xa0\x0e\x24\x8e\x04\x08\xf6\xad\x0b\x08\xef\xbe\xad\xde
\x20\xe7\x04\x08";
    char buffer[4];
    strcpy(buffer,rop);
}

int main()
{
    verify();
}
```

2) The ROP chain that we had implemented in the above program to print Yes and then jump to print_secure reworks the control flow in the following way:-

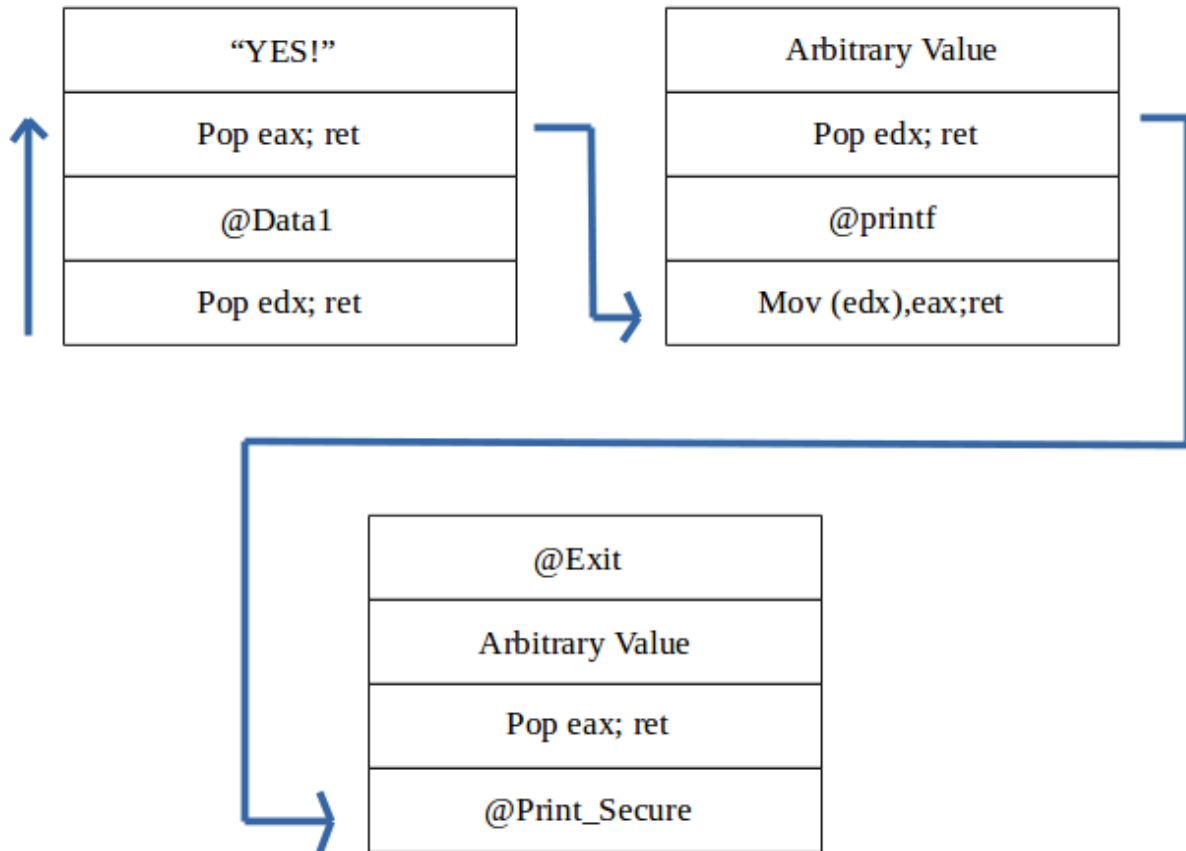


Figure 11

3) We need to find the addresses for the gadgets we have used in the rop chain which we will be doing so using ROPGadget, an open source gadget finder. You can use other gadget finders as well.

```

ROP chain generation
=====

- Step 1 -- Write-what-where gadgets

[+] Gadget found: 0x8054a42 mov dword ptr [edx], ecx ; ret
[+] Gadget found: 0x806e88a pop edx ; ret
[+] Gadget found: 0x806e8b1 pop ecx ; pop ebx ; ret
[-] Can't find the 'xor ecx, ecx' gadget. Try with another 'mov [r], r'

[+] Gadget found: 0x809a1bd mov dword ptr [edx], eax ; ret
[+] Gadget found: 0x806e88a pop edx ; ret
[+] Gadget found: 0x80bae66 pop eax ; ret
[+] Gadget found: 0x80542b0 xor eax, eax ; ret
  
```

Figure 12

4) Once you have the gadget addresses place these correct addresses in the ROP chain of the modified program and then generate the assembly level of the code using -S compiler flag. In the assembly level code, place the checksum guard template mentioned before in accordance with the number of guards you want to put. Using gdb debugger find out the correct key values and place them in the assembly level code

6) In order to ensure no one tampers with the keys you have put in the assembly level code, you can put XOR testers whose template is given below to ensure no one tampers with the keys.

```
movl    .key,%edi
xorl    .test,%edi
cmpl    %edi,.ans
je      guard
addl    %ebx,%ebp
guard:
```

Figure 13

.key => the pre-calculated key for a given piece of code
.test => an arbitrary value with which the XOR of .key is calculated
.ans => The correct XOR encrypted value.

If .key XOR .test comes different than .ans that means the precalculated key has been tampered with.

7) Finally create the executable and now you have a tamper-proof executable protected by ROP and Code Checksumming

5. Performance analysis

Code Profiling is a form of dynamic programming analysis that measures the performance of a program including the time taken by it to execute, the resources used by it etc. There are generally two ways in which profiling is done which are listed below.

1) Statistical Profiler- This kind of profilers operate by sampling. A sampling profiler samples the program's program counter at regular intervals as decided by the user based on which it does the analysis of the program.

2) Instrumentation Profiler- This kind of profiler works by adding it's own code to the executable to be profiled in order to conduct it's analysis. An example of this kind of profiler is the gprof profiler that can be used for profiling C programs.

In order to carry out profiling using this profiler the following steps need to be followed:-

- Compile the program with the `-pg` compiler flag which enables the executable to be profiled by the gprof profiler. Once that is done, execute the program to generate a `gmon.out` which will be used for our analysis

```
prashast@prashast-X550LD:~$ gcc -o profile profile.c -pg
prashast@prashast-X550LD:~$ ./profile
Initial value in Main function= 0
Last value in Main function (value(p) incremented for) = 10000
Func1(),Func2(),Func3() & filegen() called for 399 399 9601 1197 times
```

- Once that is done you can generate a flat profile or a call graph using `-p` and `-q` flag respectively

```
prashast@prashast-X550LD:~$ gprof -p profile gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
%   cumulative   self           total
time  seconds    seconds   calls  us/call  us/call  name
95.59      1.11      1.11      9601    115.49    115.49  func3
 4.34      1.16      0.05      399     126.32    151.58  func1
 0.87      1.17      0.01     1197      8.42      8.42   filegen
 0.00      1.17      0.00      399      0.00     25.26  func2

%
time      the percentage of the total running time of the
          program used by this function.

cumulative
seconds   a running sum of the number of seconds accounted
          for by this function and those listed above it.

self
seconds   the number of seconds accounted for by this
          function alone. This is the major sort for this
          listing.

calls      the number of times this function was invoked, if
          this function is profiled, else blank.

self
ms/call    the average number of milliseconds spent in this
          function per call, if this function is profiled,
          else blank.

total
ms/call    the average number of milliseconds spent in this
          function and its descendents per call, if this
          function is profiled, else blank.

name       the name of the function. This is the minor sort
          for this listing. The index shows the location of
          the function in the gprof listing. If the index is
          in parenthesis it shows where it would appear in
          the gprof listing if it were to be printed.
```

Flat Profile

```
prashast@prashast-X550LD:~$ gprof -q profile gmon.out
Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.86% of 1.17 seconds

index % time    self  children    called    name
[1]  100.0      0.00    1.17      9601/9601    main [1]
      1.11      0.00      399/399      func3 [2]
      0.05      0.01      1197/1197    filegen [4]
-----
[2]   94.8      1.11      0.00      9601/9601    main [1]
      1.11      0.00      9601          func3 [2]
-----
[3]    5.2       0.05      0.01      399/399      main [1]
      0.05      0.01      399          func1 [3]
      0.00      0.01      399/399      func2 [5]
-----
[4]    0.9       0.01      0.00      1197/1197    func2 [5]
      0.01      0.00      1197          filegen [4]
-----
[5]    0.9       0.00      0.01      399/399      func1 [3]
      0.00      0.01      399          func2 [5]
      0.01      0.00      1197/1197    filegen [4]
-----

This table describes the call tree of the program, and was sorted by
the total amount of time spent in each function and its children.
```

Call Graph



Considering how my security measures prevent the original code to be tampered with, the executable would simply break if we use an instrumentation profiler to analyze the performance of the code. Therefore, in order to analyze the performance of the code I had to use a non-intrusive approach in the form of a Statistical profiler. The statistical profiler which was used to carry out the analysis was the Intel Vtune Profiler.

In order to test the overhead of implementing these security measure on top of programs, I used a simple dummy program whose basic function was to eat CPU cycles and served as a good test case for profiling.

The profiler which I used was the Intel Vtune profiler. Before you can use the Intel Vtune Profiler to analyse the performance of your security measures, there are certain symbol which need to be linked to the executable apart from the usual compiler flags in order to ensure the Vtune is able to profile the executable.

```
gcc -fno-stack-protector -static -m32 -g -c -Wl,-u__errno_location profile1.s -o profile1.o  
gcc -fno-stack-protector -static -m32 -g -Wl,-u__errno_location profile1.o -o profile1
```

Below are the performance results of an unprotected executable versus a protected executable. The executable being the CPU cycle eating dummy program.

 Elapsed Time: 34.927s	 Elapsed Time: 34.984s
<u>CPU Time:</u> 0.550s	<u>CPU Time:</u> 0.538s
<u>Total Thread Count:</u> 1	<u>Total Thread Count:</u> 1
<u>Paused Time:</u> 0s	<u>Paused Time:</u> 0s
<i>Unprotected Program</i>	<i>Protected Program</i>

As you can see from the above comparison, the overhead of putting my security measures on top of the code is minimal and in this case there was a minimal overhead of 0.012 sec. In this case, I was only protecting the main function with a 2-guard cross verifying network. Needless to say, the overhead will increase if the number of guards or the protected code region is increased.

6. Implementation of security on an IoT testbed

In order to test my security measures in a more real life scenario, I put them on programs running on a natively developed IoT testbed. Before we get into the analysis of the performance of our code integrity measures on top of the programs running on the IoT testbed, I would like to elaborate a little bit more on the testbed that I have implemented my security measures on.

6.1 The IoT node- Intel Galileo Board

The Internet of Things (IoT) is the network of physical objects or "things" embedded with electronics, software, sensors, and network connectivity, which enables these objects to collect and exchange data. An IoT node represents the physical object which has been embedded with the above mentioned paraphernalia. In our case, the objects which were acting as IoT nodes were the Intel Galileo Boards.

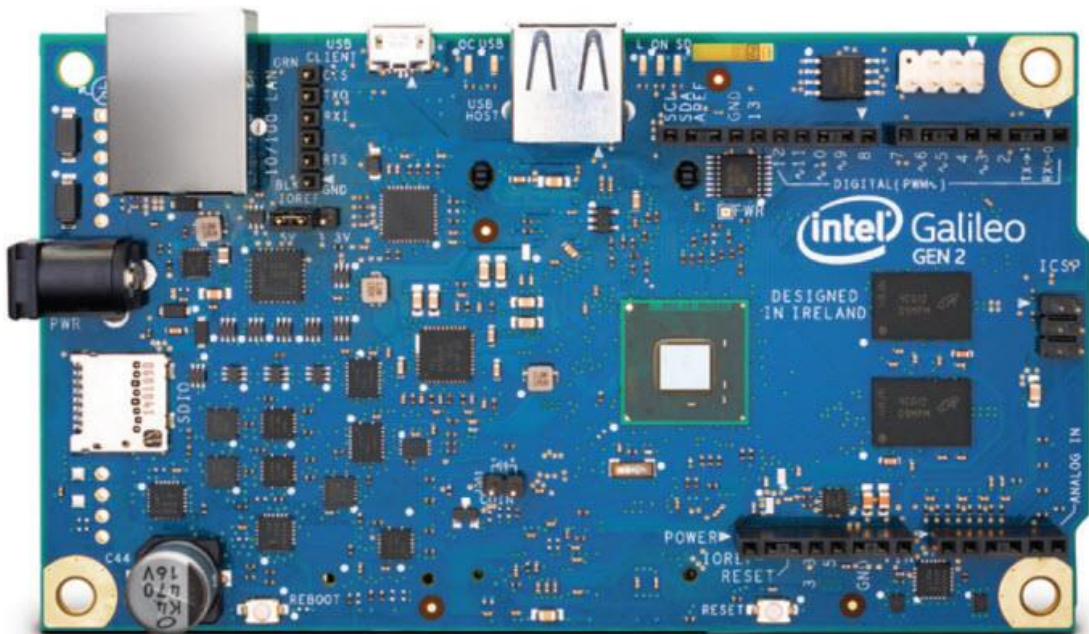


Figure 14

Some of the features of the board are as mentioned below.

- Intel Quark 32-bit processor
- Support for Linux/Windows host OS
- Compatible with Arduino Software Development environment
- 100 Mb Ethernet port

One of the major reasons this board was used to develop the IoT testbed was the freedom provided by the board in the form of a running embedded Linux OS on top of the board and SSH connectivity as well. This allowed me to not be totally dependent on IDE's such as Arduino and Eclipse to run the tamper proof code.

6.2 The IoT testbed

The testbed was serving the purpose of calculating light and temperature readings in different areas of the campus, sending them to a server through a gateway. The server constantly updates data on a webpage and alerts anyone logged onto the page if the temperature reading is above or the light reading is below a threshold value.

The networking structure of the testbed is as represented below:

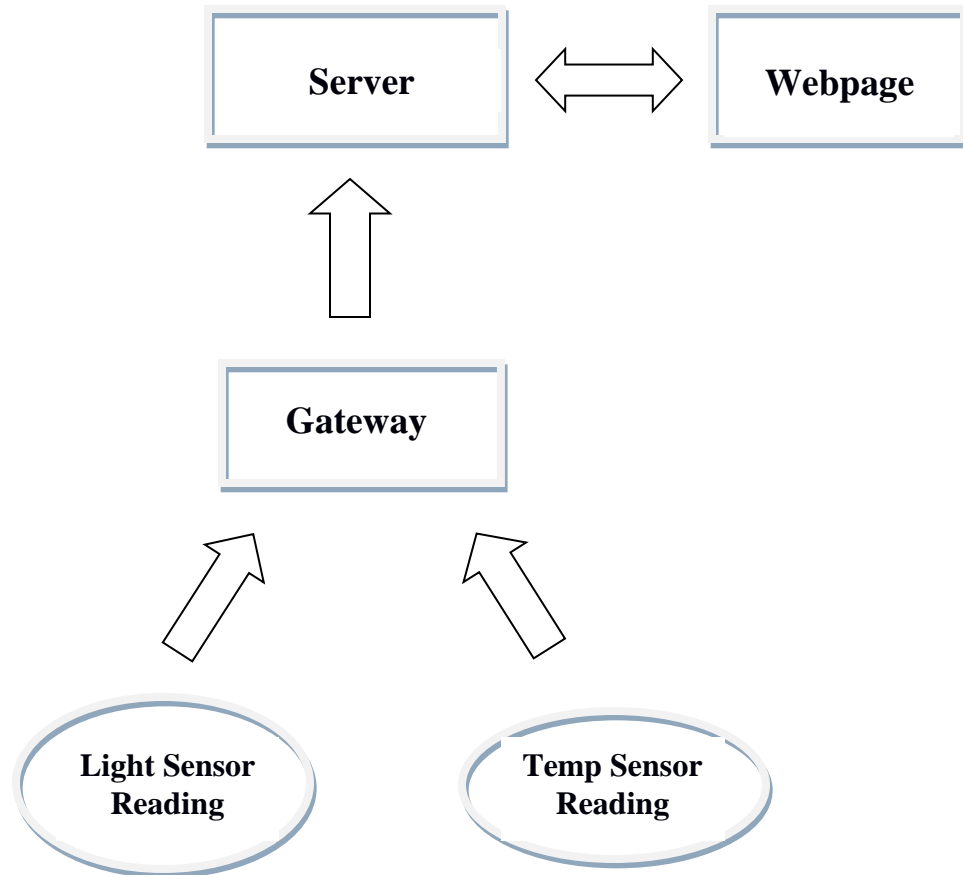


Figure 15

As you can see in the networking structure above, the light sensor and temperature sensor readings are calculated by the Intel Galileo boards serving as IoT nodes. From there, these readings are sent to the gateway which can be a workstation, in our case it was another Intel Galileo Board which was serving as Gateway.

The Gateway collects both reading periodically, concatenates them together into a single message and sends it to the Server which is responsible for updating the readings on the webpage and alerting the user logged onto the webpage if the light readings go below a certain threshold or if the temperature readings are above a certain threshold.

6.3 Methodology for running security mounted code on IoT's

In order to mount our security measures on top of the code which is running on the IoT nodes we have to make certain changes in the assembly level code which is generated. Doing this is not possible if we use an IDE such as the Arduino or the Eclipse. Eclipse does allow assembly level debugging but doesn't allow us to make any changes in the assembly code that is generated.

Therefore, in order to mount our security measures we will have to use a different approach for running our code. Because the Intel Galileo Board comes with the support for running an embedded Linux OS to which we can SSH into, there are two methods which can be used to run our security measure implemented code on the boards.

Before you can use SSH to log into the board's operating system and run your code, you need to know the IP address that has been assigned to your board by the local router. In order to do that, there are certain steps you will need to follow:-

- Enable the Ethernet capability on your board using an Arduino sketch provided by Intel itself
- Once that is done, find out the IP address which has been assigned to your board using the following sketch

- When you run the above sketch, you should be able to see the IP address assigned to the board using the serial monitor

Once you have the IP address of the board, you can proceed with the following two methods listed to run your security measures mounted code.

6.3.1 Compiling code on the board

Because the embedded Linux OS that is supported by the Intel Galileo board comes with GCC/G++ compiler along with a debugger as well, one of the alternatives which can be followed is:

- Move the code to be protected through scp file transfer to the board
- Log into the board's Linux using ssh and follow the steps according to the algorithm for implementing ROP and code checksum
- After making the appropriate changes, compile the code on the board itself and execute it over there

```
prashast@prashast-X550LD:~$ scp Read_Light_rop.c root@172.16.3.253:/home/root
```

```
prashast@prashast-X550LD:~$ ssh root@172.16.3.253
root@galileo:~# ./Read_Light_Crop
```

6.3.2 Compiling code on the workstation (PC)

This method requires a cross compilation toolchain to be setup on your workstation. An Intel Galileo board has a Quark Processor and therefore it needs code to be compiled for that processor specifically. If I compile my code on the workstation and try to run it on the board, it won't work because the code has been compiled for the processor that is there on my workstation and not for the Galileo board.

This is where the utility of the cross compiler toolchain lies. Using this toolchain I can compile code for the Quark processor on the workstation itself without me having to compile the code on the board. Therefore the steps which will need to be followed when cross compiling for the Quark Processor are:-

- Source the cross-compilation environment using the following command
- Once that is done use the following to make a cross compiled executable for your Quark processor

```
source /opt/iot-devkit/1.6.1/environment-setup-i586-poky-linux
```

```
${CC} -fno-stack-protector -g -static-libgcc -m32 -S finaltestiot1.c -o finaltestiot1.s
(Make the necessary changes in the source code)
```

```
${CC} -fno-stack-protector -static-libgcc -g -m32 -c finaltestiot1.s -o finaltestiot1.o
```

```
${CC} -fno-stack-protector -static-libgcc -g -m32 finaltestiot1.o -o finaltestiot2
```

Note that, all the commands for making the cross compiled executable should be done in a separate terminal as sourcing this environment breaks certain utilities.

6.4 Penetration Testing

In context of the structure of the IoT testbed setup as above, I mounted the security measures on top of the following layers of the testbed.

- Programs running on the Intel Galileo Boards which were functioning as the IoT nodes for calculating the light and temperature readings.
- The Gateway was also protected which was responsible for concatenating the readings and maintaining a constant uniform data flow to the server

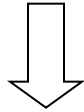
I couldn't mount the security measures on top of the server layer as it was written using Node.js which is based on the JAVA framework. Other methods would have to be used to protect this layer.

In order to test how effective was the code security measures that had been implemented against an attacker who wanted to insert malicious code to change the functionality of the executable, I tested the effect of code injection at three different points in the program along with the case if someone comes in and tampers with the checksum key as well. The results of the penetration testing are further elaborated below in the form of four test cases.

6.4.1 Inserting code within the protected code region

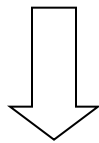
```
root@galileo:~# ./Read_Light_Crop
Light intensity is - 29.13446 lux
Temperature is - 38.47656 degree celsius
```

Untampered Executable



```
client_start:
movw    $0, -12(%ebp)
subl    $12, %esp
pushl   $0
call    mraa_aio_init
addl    $16, %esp
movl    %eax, -16(%ebp)
subl    $12, %esp
pushl   $2
addl    $1,%edi    #tampering instruction
call    mraa_aio_init
I Read_Light_Crop.s [Modified] 108/279 38%
```

**Tampering within the
protected code region**



```
root@galileo:~# ./Read_Light_Crop
Segmentation fault
```

Error occurs

As you can see from the above diagrammatic representation, if an attacker comes in and tries to make changes within the code region which is protected by code checksum, our checksumming module raises an error and tampering is detected successfully.

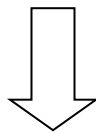
6.4.2 Inserting code outside the protected code region

As discussed before the code checksumming module protects a region of code as specified by the user using labels. Through experimental analysis, it has been found out that the protection extends even outside the region of code which is explicitly protected by the code checksum procedure. More specifically, the region of code which has been specified before the functions in which the code checksum has been activated.

The reason behind why this happens is that there are certain jump instructions which use the labels as addresses to jump. If an attacker comes in and makes some changes in the region of code above where the code checksum guards are specified, the addresses of these labels change and indirectly we get to know that tampering has occurred because our checksum module raises an error.

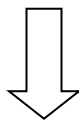
```
root@galileo:~# ./Read_Light_Crop
Light intensity is - 29.13446 lux
Temperature is - 38.47656 degree celsius
```

Untampered Executable



```
.file "Read_Light_Crop.c"
.text
.globl main
.type main, @function
main:
.LFB2:
.cfi_startproc
leal 4(%esp), %ecx
.cfi_def_cfa 1, 0
andl $-16, %esp
pushl -4(%ecx)
pushl %ebp
.cfi_escape 0x10,0x5,0x2,0x75,0
movl %esp, %ebp
addl $1,%edi #tamper
pushl %ecx
.cfi_escape 0xf,0x3,0x75,0x7c,0x6
subl $4, %esp
call verify
addl $4, %esp
popl %ecx
.cfi_restore 1
.cfi_def_cfa 1, 0
I Read_Light_Crop.s [Modified] 17/280 6%
```

Tampering outside the protected code region



```
root@galileo:~# ./Read_Light_Crop
Segmentation fault
```

Error occurs

6.4.3 Inserting code in a different linked C program

In the context of my test case, I was supplied with two C programs:- Light_Client.c and Read_Light.c. Light_Client.c contained the definition of the functions which were being used in the Read_Light.c to read Light and temperature readings. So an executable was made with these two programs linked together.

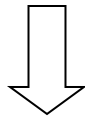
I had mounted my dual security measures of ROP and Code Checksum on the Read_Light.c program considering how this contained the code which was responsible for communicating with the sensors and relaying the data to the gateway.

It was experimentally found out that even though I hadn't mounted any security measures explicitly on top of the Light_Client.c program, the security measures implemented in the other program, more specifically the ROP module provided security to the code written in Light_Client.c

The reason behind this is that the gadget list which we find from the resulting executable and use in our ROP chain contains gadgets both from Light_Client.c and Read_Light.c. Therefore, if an attacker comes in and tries to tamper with the definitions of the functions in the Light_Client.c, the resulting addresses of the gadgets in the tampered executable change and our ROP module specified in the other program is able to identify that the executable has been tampered with.

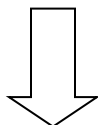
```
root@galileo:~# ./Read_Light_Crop
Light intensity is - 29.13446 lux
Temperature is - 38.47656 degree celsius
```

Untampered Executable



```
int open_connection_to_gateway(){
    int sockfd, portno;
    struct sockaddr_in serv_addr;
    struct hostent *server;
    portno = 50000;
    printf("1\n"); //tampering instruction
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");
    server = gethostbyname("172.16.3.156");
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }
    bzero((char *) &serv_addr, sizeof(serv_addr));
I light_client 1.c [Modified] 29/44 65%
```

Tampering in the linked C program



```
root@galileo:~# ./Read_Light_Crop
Segmentation fault
```

Error occurs

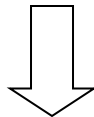
6.4.4 Tampering with the checksum keys

In my implementation of code checksum, the correct checksum values (keys) have been specified in the assembly code itself. Therefore, there is a possibility that if an attacker identifies these keys and the regions to which those keys correspond to, he can simply change the values specified as keys according to his liking.

In order to plug this vulnerability in my implementation, I had spread XOR testers as discussed before throughout the code at arbitrary locations. The function of the XOR testers is simply to ensure that the correct checksum values hadn't been tampered with at all.

```
.key:
    .long 0xce916390
.key1:
    .long 0xd5f79b56
.test:
    .long 0xdeadbeef
.test1:
    .long 0xaabbccdd
.ans:
    .long 0x103cdd76
.ans1:
    .long 0x7f4c578b
.LC10:
    .string "Temperature is - %.5f degree celsius\n"
    .text
    .globl rop1
```

This key has been tampered with



```
root@galileo:~# ./Read_Light_Crop
Segmentation fault
```

Error occurs

In this case the .key value had been tampered with. Once that happens, the .key XOR .test leads to a different result other than that specified in .ans. Since that happens, the XOR tester gets to know that the key has been tampered with, corrupts the base pointer and raises a segmentation fault.

7. Scope for future work

There is lot of scope for future work ahead. The possible ways in which this work can be further researched upon has been provided below.

- In my current implementation, the checksum guards have to be placed manually. With an increasing number of guards, the complexity related with placing the guards such that there are no clobbered registers increases exponentially. Therefore, an automated approach has to be developed for the placement of the guards according to the number of guards that the user wants to use to create a strongly connected network
- The ROP chain that I am making right now has been done manually using the gadget finder to find the gadget addresses. As such, the functionality by these manually created chains is very simplistic. In order to create more complex chains, we will need to integrate Parallax, a ROP chain creator which has been developed by our collaborators at VU University, Amsterdam and incorporate that prototype into our current security implementation
- While searching for gadgets in an executable that had been cross compiled for the Quark processor, it was observed that the quality of gadgets is abysmal with a very less gadgets being found throughout the executable. Therefore, we need to find a way to insert gadgets so as to improve the quality of the gadgets and build a more complex ROP chain that covers relatively more of code region when we are mounting these security measures on top of code running on IoT nodes
- In my current implementation, the regions which were to be protected were decided based on the CPU usage and by doing a hotspot analysis using a profiler. In the context of IoT nodes, the hotspots that utilize maximum CPU time aren't the only regions which need to be protected. In addition to that, there are security vulnerable code regions, for eg. The code chunks responsible for relaying data from one layer to another or the code chunks which are reading sensor data and doing the necessary manipulations. If a malicious attacker is able to come in and tamper with these kind of code regions, the results could be catastrophic. Therefore, a profiler is required to identify such points of vulnerability as the current profilers only are able to identify hotspots and not security vulnerable code.

8. Conclusion

This undergraduate thesis was aimed at developing a lightweight method of code self-verification for IoT devices. By the end of the period of this thesis, I have been able to develop a lightweight code self-integrity measure in the form of a dual implementation of Return Oriented Programming along with Code Checksum.

The strength of this security measure lies in the fact that these two security modules i.e. ROP and Code Checksum aren't acting in isolation with each other but in fact they act in tandem, complimenting each other.

Therefore, the only way an attacker can come in and tamper with the critical regions of code is if he simultaneously disables both the security modules. This technique is serving the purpose of hardening the security. The amount of effort and computation power that the attacker will have to put in in order to bypass these security measures will be of high magnitude sometimes not even commensurate with the returns he will get from tampering with the software. This will serve to deter the attackers from mounting an exploit and tampering with the code

References

- [1] H. Shacham, “The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86),” in Proceedings of the 14th ACM conference on Computer and Communications Security (CCS) , 2007.
- [2] D. Andriessse, H. Bos, and A. Slowinska, “Parallax: Implicit Code Integrity Verification Using Return-Oriented Programming”, in Proceedings of the 45th Conference on Dependable Systems and Networks (DSN'15), (Rio de Janeiro, Brazil), IEEE Computer Society, June 2015.
- [3] H. Chang and M. Atallah. Protecting software code by guards. In Proc. 1st ACM Workshop on Digital Rights Management (DRM 2001), volume 2320 of Lecture Notes in Computer Science, pages 160-175. Springer-Verlag, 2002.
- [4] G. Wurster, P. van Oorschot, and A. Somayaji. A Generic Attack on Checksumming-Based Software Tamper Resistance. In Proceedings of IEEE Symposium on Security and Privacy, S&P'05, 2005.

Lightweight Code Self Verification for Internet of Things(IoT) devices

ORIGINALITY REPORT

3%	1%	2%	%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Andriesse, Dennis, Herbert Bos, and Asia Slowinska. "Parallax: Implicit Code Integrity Verification Using Return-Oriented Programming", 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2015. Publication	1%
2	en.wikipedia.org Internet Source	<1%
3	socialmediaforlearning.com Internet Source	<1%
4	Pantilimonescu, Florin, Lucian Constantin Hanganu, Mihaita Peptanariu, Stefan Grigoras, Irina Ionescu, Georgeta Lidia Potop, Alina Iovan-Dragomir, and Stela Carmen Hanganu. "Modular Student Learning Kit for Internet of Things", Applied Mechanics and Materials, 2014. Publication	<1%

5	www.3g.co.za Internet Source	<1%
6	www.abica.co.uk Internet Source	<1%
7	www.seediscover.com Internet Source	<1%

EXCLUDE QUOTES ON

EXCLUDE MATCHES OFF

EXCLUDE
BIBLIOGRAPHY ON