BUFFER OVERFLOW ATTACK MITIGATION VIA TRUSTED PLATFORM MODULE (TPM)

Teh Jia Yew[1], Khairulmizam bin Samsudin[2], Nur Izura Udzir [2] and Shaiful Jahari bin Hashim[2]

[1]*Infrastructure University of Kuala Lumpur, Malaysia*
[2]*Universit Putra Malaysia, Malaysia*

ABSTRACT
As of the date of writing of this paper, we found no effort whatsoever in the employment of Trusted Computing (TC)'s Trusted Platform Module (TPM) security features in Buffer Overflow Attack (BOA) mitigation. Such is despite the extensive application of TPM in providing security based solutions, especially in key exchange protocols deemed to be an integral part of cryptographic solutions. In this paper we propose the use of TPM's Platform Configuration Register (PCR) in the detection and prevention of stack based buffer overflow attacks. Detection is achieved via the integrity validation (of SHA1 hashses) of both return address and call instruction opcodes. Prevention is achieved via encrypting the memory location addresses of both the return and call instruction above using RSA encryption. An exception is raised should integrity violations occur. Based on effectiveness tests conducted, our proposed solution has successfully detected 6 major variants of buffer overflow attacks attempted in conventional application codes, while incurring overheads that pose no major obstacles in the normal, continued operation of conventional application codes.

Keywords:
Buffer Overflow Attack, Trusted Platform Module, Platform Configuration Register (PCR), ptrace, TPM_Extend, TPM_Seal.

INTRODUCTION

Buffer Overlfow Attacks (BOA) appears as of the date of writing of this paper- found in leading exploit reporting sites such as packetstorm security and exploit-db [1, 2]. Such is despite BOA was initially first reported by Aleph One way back in 1996 [3]. Untill BOA can be eradicated to an appreciably safe level or eradicated, work into BOA mitigation shall further necessitates momentum.

The source of BOA lies in the absence of limit check in codes, resulting in overflowing of statically allocated buffers. A simple example to illustrate BOA is via program tesbuff.c below [4]. Note that tesbuff.c and program refers to the same entity throughout this paper.

```
#include <unistd.h>

void Tesbuff()
{
char buff[4];
printf("Some input: ");
gets(buff);
puts(buff);
}

int main(int argc, char *argv[ ])
{
Tesbuff();
return 0;
}
```

Figure 1: Tesbuff.c – A Sample Code Vulnerable to BOA.

Input of chars in excess of four overflows buff array and results in the overwriting of the return address (0x40253f in Figure 2 below) of the main function in the memory stack. Overwriting of the return address results in the existence of vulnerability for exploit deployment, an example of one of such exploit is the spawning of a shell (via shellcodes) - a form of privilege escalation attack where control is redirected from the compromised host to a malicious adversary [6].

Some notable work focuses only on return address vulnerability attack mitigation were: hardware based protection of the return address as per SmashGuard [7], a micro-architecture based approach as per Park et al. [8] and compile time protection of return address as per Return Address Defense (RAD) [9].

Another aspect of an attack on a program's memory stack is the modification of the call instruction opcode (see 0x40253a in Figure 2 below) to effect a control flow hijacking attack. It was demonstrated via Return Oriented Programming (ROP) attacks that unintended instruction sequences can be introduced into x86 instruction sequences in the memory stack, particularly over the opcodes of an instruction. Such unintended instructions were never intended by the program or compiler [5]. Consider the following gdb dissassembly of the buffer overflow vulnerable code testbuff.c in Figure 1 above.

```
(gdb) disas main
Dump of assembler code for function main:

0x0000000000402526 <+0>: push   %rbp
0x0000000000402527 <+1>: mov    %rsp,%rbp
0x000000000040252a <+4>: sub    $0x10,%rsp
0x000000000040252e <+8>: mov    %edi,-0x4(%rbp)
0x0000000000402531 <+11>:  mov    %rsi,-0x10(%rbp)
0x0000000000402535 <+15>:  mov    $0x0,%eax
0x000000000040253a <+20>:  callq  0x4024f0 <Tesbuff>
0x000000000040253f <+25>:  mov    $0x0,%eax
0x0000000000402544 <+30>:  callq  0x4024e0 <Display>
0x0000000000402549 <+35>:  mov    $0x0,%eax
0x000000000040254e <+40>:  leaveq
0x000000000040254f <+41>:  retq
```

```
End of assembler dump.

(gdb) disas Tesbuff
Dump of assembler code for function Tesbuff:
0x00000000004024f0 <+0>: push   %rbp
0x00000000004024f1 <+1>: mov    %rsp,%rbp
0x00000000004024f4 <+4>: sub    $0x10,%rsp
0x00000000004024f8 <+8>: mov    $0x408ca5,%edi
0x00000000004024fd <+13>:  mov    $0x0,%eax
0x0000000000402502 <+18>:  callq  0x401e20 <printf@plt>
0x0000000000402507 <+23>:  lea    -0x10(%rbp),%rax
0x000000000040250b <+27>:  mov    %rax,%rdi
0x000000000040250e <+30>:  mov    $0x0,%eax
0x0000000000402513 <+35>:  callq  0x401e50 <gets@plt>
0x0000000000402518 <+40>:  lea    -0x10(%rbp),%rax
0x000000000040251c <+44>:  mov    %rax,%rdi
0x000000000040251f <+47>:  callq  0x4022c0 <puts@plt>
0x0000000000402524 <+52>:  leaveq
0x0000000000402525 <+53>:  retq
End of assembler dump.
```

Figure 2: gdb Disassembly of Tesbuff.c Memory Stack.

The control flow hijacking of the call instruction manifests when the original call was replaced with something like:

```
callq  0x4024fe <Evil_Tesbuff>
```

Program execution flow would then be redirected to the malicious function Evil_Tesbuff. Any malicious code residing in Evil_Tesbuff can subsequently be deployed.

While some work exists on protecting return address as mentioned earlier, we had discovered that till date no attempt was made to leverage any form of protection on the call instruction in memory stack from such control flow redirection vulnerability. While the return address is secured, the call instruction is still vulnerable [7-9]. Hence, the effective mitigation measures of BOA necessitates the protection of both the return address and call instruction opcodes. We shall refer to both to be Vulnerable Entry Points (VEP) throughout this paper.

Reasonably recent efforts into BOA mitigation proposed the vulnerability patching method: upon program binary execution, compromised vulnerabilities in overflowed programs are automatically patched to restore the program into normal working order and vulnerable buffers are moved into protected memory regions. Examples of such approaches: SafeStack and SoupInt [11, 12]. The patching methods employed, while effective in plugging BOA vulnerabilities, suffer from a major drawback: prevention occurs after damage/upon had been done. We are of assertion that an improved methodology would be to permit complete program execution only after the binary passed all essential integrity validations, especially at the vulnerable return and call locations.

In an attempt to propose a viable solution towards: a) the protection of both return address and call instruction opcode, b) prevention of the exploitation of BOA vulnerability prior to complete program execution, we propose the design and implementation of our solution: hardware anchored mitigation of BOA via Trusted Platform Module (TPM). We are the first to utilise TPM in BOA mitigation work, as despite being first introduced in 2002, no effort was found to leverage TPM security features into BOA mitigation work.

Our approach utilises TPM security features of TPM_Extend and TPM_Seal into the two main arms of our BOA mitigation solution– the Validation and Preventive Modules, (VM) and (PM) respectively [13]. We begin with an initial state whereby the program is initially in a non operative RSA encrypted state, starting from location 0x40253a onwards (i.e. the Vulnerability Point or VP). The private RSA key is sealed into TPM hardware registers – the Platform Configuration Registers (PCR) for tamper resistance. Decryption (unseal) of the VP location (i.e. lifting of non-operative state) is permitted upon successful validation of the hashes (clean vs. runtime) of both return address and call instruction opcode – by the VM. Both hashes are stored (or extended) into TPM PCR – again for tamper resistance.

The contributions of our paper are as follows:

a. we had proposed an improved solution of BOA mitigation encompassing two major vulnerable points in a binary: return address and call instruction (which has so far being neglected in BOA mitigation works). BOA mitigation work providing solution at return address targets only the return address.

b. our proposed solution is hardware based – hence tamper resistant and its integrity is guaranteed. Such is due to the fact that the Validation Gadgets [(VG) – (See next section)] - are stored in hardware registers i.e. PCRs – the theft of which is impossible via software based attacks.

c. we had, to date, implemented the VM in 64-bit Fedora Linux OS and evaluated its effectiveness and performance using actual BUFFER OVERFLOW exploit codes.

Our proposed solution is anticipated to be ideal for real-world deployment due to the hereinlisted strengths:

a. Acceptable systems overhead
Hashes of VG are generated using TPM hardware SHA1 engine, hence we anticipate there will be acceptable penalty in terms of consumption of system resources.

b. Total trustworthiness
Absolute Integrity of the employed BOA Detection Mechanism is guaranteed due to TPM anchored protection of VG. Total trust can be placed on our solution since there's no way VG hashes can be stolen from TPM hardware registers.

The rest of the paper is organised as follows: Section 2 details the design and architecture of our proposed solution, Section 3 details the methodology employed in implementing our solution, Section 4 provides the experimental results and finally Section 5 summarises our work.

I. DESIGN

We introduce a high level overview for the design of our solution in this section. We then elaborate on how each component of our system functions in BOA mitigation. Our solution is based on the idiom 'its better to look before you leap' – thus VG shall be deemed to be malicious until proven otherwise. As mentioned in Section 1, our solution comprises two major arms- the VM and PM.

Note that the non operative state is the first tentative step towards realizing the idiom above. The PM's refined operating principle employed is that both memory locations and instruction opcodes are deemed to be malicious unless proven otherwise.

The initial state of the program binary is enforced by the PM, serving as an Enforcement Mechanism. All operations in memory addresses beginning from the call instruction to the bottom of the program's memory stack (i.e. from 0x40253a onwards) shall initially be in a non operative state. To achieve such a state, memory addresses from 0x40253a onwards need to be in a RSA encrypted state (whereby the private RSA key is sealed to TPM PCR using TPM_Seal).

The non operative state shall be lifted (i.e. memory addresses decrypted or unsealed using TPM_Unseal) only upon Validation Gadgets (see paragraph below) passing the validation stage by the VM as per next paragraph below.

Next, the VM serves as an Integrity Checkpoint Mechanism – verifying the integrity of Validation Gadgets (VG) which comprises both hashes (of return address and call instruction opcodes) and RSA private encryption keys. Note that the VGs are extended (i.e. stored using TPM_Extend) in TPM PCR for tamper proof measures, which in turn guarantees the integrity both of the VM and PM, i.e. guards the guard.

In the Validation Module, which serves as an Integrity Validation Checkpoint, the refined operating principle employed is that the program shall pass integrity validation checks at points of vulnerability.

We thus utilise the TPM built in SHA1 hash engine to generate both clean and runtime hashes of a program's function return address and call instruction opcode. The hashes of an uninfected program's return address and the call instruction opcode are extended into two PCRs (PCR - 13&14). See Figure 6. The runtime hashes of the identical program's return address and call instruction opcode are compared via an Integrity Assessment Engine (IAE). Mismatch indicates abnormal program behavior – hence the program is un-trustable.

II. IMPLEMENTATION

This section details the implementation of both the VM & PM in our proposed solution. Our solution mechanism comprises three phases: Pre-Deployment, Deployment and Post Deployment.

A. Pre-Deployment Phase

The internal architecture of the VM is illustrated in Figure 3 below. The core components of the VM are the Checkpoint Trap (CT) , PCR Extender (PE) and the IAE.

Prior to the deployment of the VM, a clean database of the SHA1 of the VG needs to be made available for a runtime comparison of the VG of the testbuff binary. The PE component utilises the TSPI compliant function Tspi_TPM_PcrExtend () to both generate the SHA1 of the VGs and then extend the VGs to a chosen TPM PCR.

The CT functions similar to the gdb debugger – by initially forking the testbuff binary as a child process (via the fork () system call) and then setting breakpoints at VEPs, hence

preparing the environment for the IAE to perform its operations of integrity assessment of the VG.

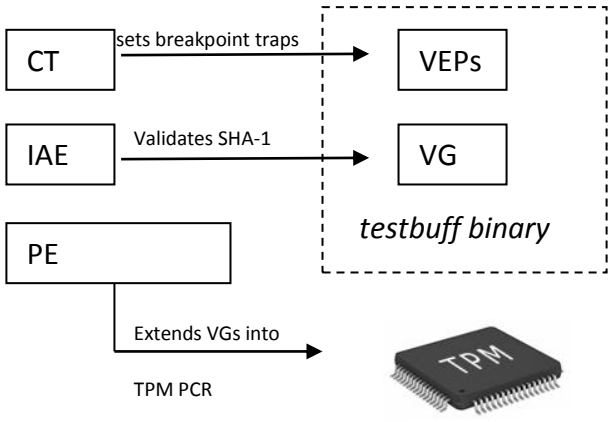

Figure 3: Architecture of the VM.

B. Deployment Phase

Breakpoints need to be set at two locations i.e. both the VEP as previously mentioned. The ptrace() system call is utilised to develop a gdb like feature – of setting breakpoints at both the VEP memory address locations [10]:

a. of the call instruction in the memory stack of a binary in execution (i.e. binary of compiled tesbuff.c) – we term this as VEP-1, and,

b. of the return address of tesbuff.c. We term this as VEP-2.

Breakpoints require the use of trap instruction - 0xcc. Hence, the VM writes the 0xcc into the opcodes of both VEP1 and VEP2 via the use of the POKE_TEXT enumeration parameter supplied to the ptrace () function. [10]. The code fragment below demonstrates the setting of a trap at VEP2:

```
unsigned return_addr = 0x000000000040252f ;  // address for
                                                VEP2

/* Write the trap instruction 'int 3' into the address in return_addr */
   unsigned ret_data_with_trap = (ret_data & 0xFFFFFF00) | 0xCC;
   ptrace(PTRACE_POKETEXT, child_pid, (void*)return_addr, (void*)ret_data_with_trap);

unsigned ret_readback_data = ptrace(PTRACE_PEEKTEXT, child_pid, (void*) return_addr,
0);

procmsg("After trap SIGNAL, data at return_addr (i.e RET ADDRESS) at 0x%08x: is ->
0x%08x \n\n", return_addr, ret_readback_data);
```

Upon code execution, the RIP (64 bit - Instruction Pointer) points to the location of the return_addr signifying that a trap had been set at 0x40252f. The terminal output (Figure 3) below shows the Instruction opcode of the return address has been changed from b8 to cc. See Figure 3 – indicated by the blue arrow. Such result in the child (testbuff binary) entering into a trap state, permitting the IEA to perform its operations of clean vs. runtime comparison of SHA-1 of VGs at VEP-1 and VEP-2.

The RIP is not permitted to 'point' to the call and return address memory locations respectively till the IAE approves of the integrity of the return address and call instruction opcode respectively.

C. Post Deployment Phase

An alert is immediate generated (see Figure 4 - red rectangle) should the hashes are not identical, signaling the occurrence of BUFFER OVERFLOW in the traced binary. Note that in Figure 3 below, the input of eight 'A's lead to the overflowing of the char buff array (see Figure 1) and hence the overwriting of the original return address at 0x040252f [4]. The PM operative state will not be lifted and the memory addresses after the 0x040252f location will remain in RSA encrypted state – such prevents the deployment of any form of exploits or malicious codes that results from the BUFFER OVERFLOW attack on the vulnerable testbuff binary.

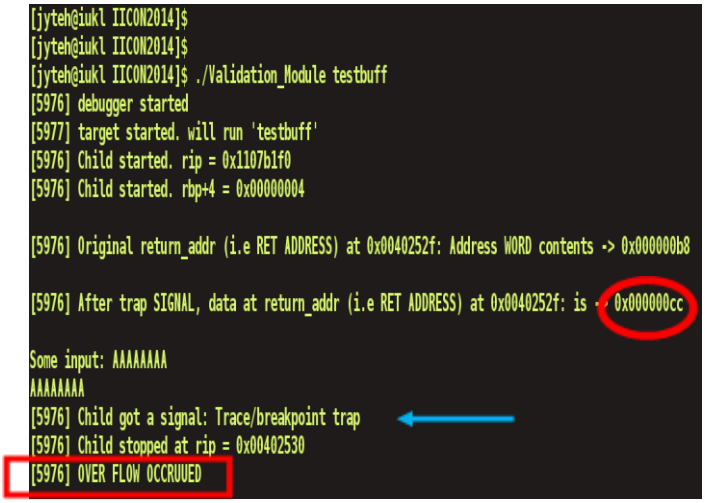

Figure 4: Validation Module in Deployment

EVALUATION

We had conducted capability tests on our proposed solution via effectiveness and performance evaluations. All experiments were conducted on an actual system for live, real time results. Our test-bed platform: Acer Veriton PC with 4GB RAM and Intel i3 Processor running Fedora Core 20 (64 bit) kernel version 3.15-6.

A. Effectiveness

We ran the VM against the list of well known BUFFER OVERFLOW attack variants as per Table 1. The modus operandi of these variants targets the function return address of programs - whereby the return address needs to be overwritten in order for the variants in Table 1 to operate. The VM is able to reliably detect overflow occurrences as shown in Table 1. The major categories of buffer overflow attack variants detected successfully by the VM demonstrates that our solution is indeed one vital addition into the current and ongoing research into buffer overflow attack mitigation.

B. Performance

To demonstrate that our solution is not only reliable but further ideal for deployment in actual environments we ran microbenchmark tests to gauge any overheads introduced by our proposed solution.

We initially measure the execution time of running the compiled C source binaries of all the buffer overflow attack variants in Table 1 without the VM and with inputs triggering overwrite of a singe return address. The execution time measurement is repeated, with the identical binaries and configurations as per the initial step but with the VM running. To obtain the required data for performance benchmarks, we utilised the most direct method for the acquisition of execution time of binaries, the time tool which operates with the gettimeoftheday() as the main software timer component. Figures 5 and 6 illustrate our performance measurement results.

Our deployment of the VM reported an average increase of ratio of binary execution time of 2.30. The ratio was due to the setting and removal of breakpoints in all the buffer overflow variant binaries in Table 1, hence resulting in the increase of binary execution time. Further, verification of the binary with the VM involves the execution of two binaries simultaneously, hence the reason behind the average execution time of 2.30 across all buffer overflow variants as per Figures 5 and 6. However, hash computation of VG were found not to contribute to the ratio increase - since the TPM's built in SHA1 engine was utilised in hash computation for the VG. Furthermore, the time tool measures execution binary time utilising only software based clocks. Hence, any lags in software execution time shall bound to be reported by the time tool.

Table 1: BOA Variants Detected by the VM

BUFFER OVERFLOW Variant No.	**BUFFER OVERFLOW Attack Variant**	**Detected by VM**
1	Local Char Array	

	Buffer Overflow (i.e. *testbuff*.c)	Yes
2	Integer Overflow	
3	Return to lib-C	
4	Format String Attack	
5	NULL Pointer Dereferencing	
6	ROP based Exploit	

Figure 5: Performance Analysis – Measurement of Binary Execution Time of Our Proposed Solution

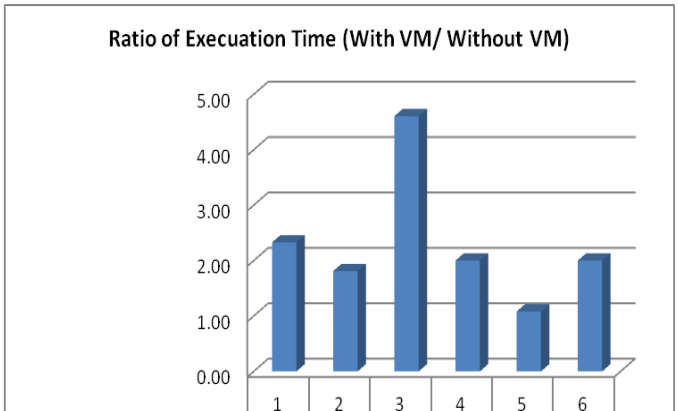

Figure 6: Ratio of Execution Time

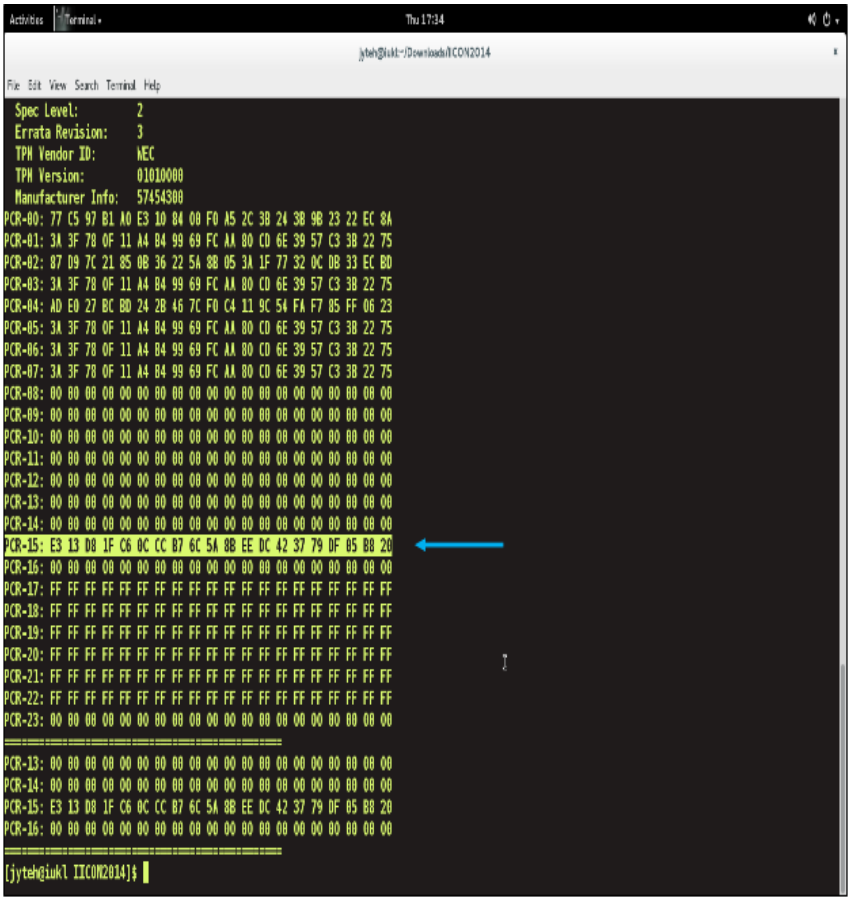

Figure 7: Extending VG into TPM PCR

CONCLUSION

We had presented a vital hardware based addition to the existing arsenal of mitigatory research efforts towards combating buffer overflow attacks, which continues to plague applications developed even till today. Our solution operates based on the principle that programs should not be granted execution privileges until its integrity had been duly verified hence ensuring that only safe, legitimate programs are permitted to be granted execution privileges.

The TPM anchored VM is guaranteed of its tamper resistance since the clean hashes of VG are stored in the PCR – the guard is totally trustable. Such is unlike past software only based buffer overflow mitigation works which posed risks of compromise – the guard isn't totally trustable. Furthermore, our solution proposed buffer overflow mitigation via two vital attack vectors in buffer overflow attack variants -both the call instructions and return addresses.

The empirical evaluation outcome of our approach i.e. a) the successful detection major buffer overflow attack variants and b) acceptable average ratio of execution time of 2.30 (with no reported major lag in binary execution upon validation by the VM – consolidates the viability of our proposed solution.

FUTURE WORK

We are in the process of adapting our proposed solution to solve the possibility buffer overflow attack on SELinux hooks (i.e function pointers) – the vulnerability in which we had previously reported in SELGuard [14]. Porting our proposed solution to SELinux requires implementation of both the VM and PM at the kernel level. Our pioneer work of buffer overflow attack mitigation with TPM anchorage opens another avenue for enhancing buffer overflow mitigation – we plan to append remote detection capabilities to the VM. Such is achieved via the use of remote attestation [15], whereby in networked systems, users can perform an attestation test to verify the authenticity and integrity of a executing program binary via clean-runtime comparison of hashes of VG.

The clean hashes of VG are stored in a remote user's database (secured with TPM) and runtime hashes of VG are delivered from the server (hosting a program binary) to the remote user. The user system performs comparison of the hashes (clean vs. runtime) the output of which will determine if a binary is executing in an untampered execution environment .We foresee such remote attestation mechanism deployable in financial business applications where integrity and trust are of vital importance – one such example is in Internet Banking.

ACKNOWLEDGEMENT

The authors wish to thank the Robotics & Embedded Systems and Information Systems Security Laboratories, Faculty of Engineering and Faculty of Computer Science & Information Technology, UPM, for technical and equipment assistances rendered and the main author further wishes to thank the Malaysian Ministry of Education (Higher Education Division) for the award of the myBrain – myPhD scholarship in support of our work.

REFERENCES

[1] Packetstorm Security Resources (2014, September 30). *HP Network Node Manager I PMD Buffer Overflow*. Retrieved from http://packetstormsecurity.com/files/128478/HP-Network-Node-Manager-I-PMD-Buffer-Overflow.html.

[2] Exploits Resources (2014, August 27). Retrieved from http://www.exploit-db.com/exploits/34421/.

[3] One, A. (1997). *Smashing the stack for fun and profit*. Retrieved from http://insecure.org/stf/smashstack.html.

[4] Tenouk. (2014, October 14). *Tutorial on Stack Based Buffer Overflows*. Retrieved from http://www.tenouk.com/Bufferoverflowc/Bufferoverflow4.html

[5] Lucas Davi et. al. (2011). ROP-defender: A Detection Tool to Defend against Return Oriented Programming Attacks. ASIACCS'11, ACM.

[6] University of Maryland Cybersecurity Lab. (2012). Shellcode and Buffer Overflow Lab. Retrieved from http://www.cs.umd.edu/class/fall2012/cmsc498L/materials/vuln-lab.shtml.

[7] Hilmi Ozdoganoglu et. al. (2006). SmashGuard: A hardware solution to prevent security attacks on the function return address. *IEEE Transactions on Computers, 55(10),* 1271-1285.

[8] Yong, J. P., & Gyungho, L. (2004). Repairing return address for buffer overflow protection. CF04, ACM, 335 – 342.

[9] Chiueh, T., & Hsu, F. (2001). RAD: A Compile-time solution to buffer overflow attacks. Proc. 21st Int'l Conf. Distributed Computing Systems (ICDCS '01), 409-417.

[10] Tutorials Point (2014, October 14). *The Ptrace () system call.* Retrieved from http://www.tutorialspoint.com/unix_system_calls/ptrace.htm.

[11] Gang, C. et.al. (2013). SafeStack – Automatically patching stack-based buffer overflow vulnerabilities. *IEEE Transactions on Dependable and Secure Computing, 10(6),* 368 – 387.

[12] Tielei, W. et al. (2014). Diagnosis and Emergency patch generation for integer exploits. DIMVA 2014, LNCS 8550, 255-275.

[13] Trusted Computing Group. (2007). TPM main part 3 commands specifications-version 1.2, p. 60, 157.

[14] The, J. Y. et al. (2013). Guarding the guard: Towards leveraging tpm based attestation on selinux security function hooks. IUKL ITInfra.

[15] University of Washington Computer Science & Engineering Dept. (2014, October 14). *Attestation and trusted computing. Retrieved from* https://courses.cs.washington.edu/courses/csep590/06wi/finalprojects/bare.pdf.