A multi-factor approach to securing software on client computing platforms

Raghunathan Srinivasan, Partha Dasgupta
Arizona State University
Tempe, USA
{raghus, partha}@asu.edu

Vivek Iyer, Amit Kanitkar
Microsoft Corp.

Sujit Sanjeev
Goldman Sachs Group Inc.

Jatin Lodhia
Google Inc.

***Abstract*—Protecting the integrity of software platforms, especially in unmanaged consumer computing systems is a difficult problem. Attackers may attempt to execute buffer overflow attacks to gain access to systems, steal secrets and patch on existing binaries to hide detection. Every binary has inherent vulnerabilities that attackers may exploit. In this paper we present three orthogonal approaches; each of which provides a level of assurance against malware attacks beyond virus detectors. The approaches can be added on top of normal defenses and can be combined for tailoring the level of protection desired. This work attempts to find alternate solutions to the problem of malware resistance. The approaches we use are: adding diversity or randomization to data address spaces, hiding critical data to prevent data theft and the use of remote attestation to detect tampering with executable code.**

Keywords—Computer security, attacks, memory randomization, secure key storage in memory, remote attestation, integrity measurement.

I. INTRODUCTION

The magnitude of the threat from malware, especially on "consumer computing" platforms is well known and well understood. Malware today can hide from virus detectors, steal secrets, live stealthily for extended periods of time, effectively prevent removal efforts, and much more. The ability to run sensitive applications and store sensitive data on consumer platforms is very critical. A smartly designed malware can have more power than any other application in the system [1]. A sensitive application running on an un-trusted environment can be subject to a variety of attacks. Attestation of client computers using hardware attestation modules, or using hypervisors to scan computers have not had much success due to the cumbersomeness of the solutions. In this paper, three orthogonal approaches are represented, each of which which provides a level of assurance against malware attacks beyond virus detectors. The approaches can be added on top of normal defenses and can be combined for tailoring the level of protection desired. This work attempts to find alternate solutions to the problem of malware resistance.

The first approach is the ability to provide "software diversity" for legacy software. Currently a malware designer can perform offline analysis of an application to discover vulnerabilities in it. These vulnerabilities can be exploited to launch various kinds of attacks on multiple systems. Every copy of an application that is shipped to consumers is exactly the same, and contains the same weaknesses in the same binary locations. Software diversity breaks up the uniformity making each instance of the application different, and attacks that work on one instance do not work on another. ASLR [2] is an example, but that idea is taken to finer degrees of granularity on each stack frame and heap frame in this work. The second approach is to build obfuscation and shielding methodology to make stealing secrets from client machines harder. For example, memory in client applications holds encryption keys, passwords, sensitive data, and private keys in case of PKI implementations. Stealing secrets by copying zones of memory is particularly simple (keys for example, have high entropy). Two approaches to hide keys more effectively are provided in this work. The third approach is Remote Attestation. Remote attestation is a set of protocols that uses a trusted service to probe the memory of a client computer to determine whether one (or more) application has been tampered with or not. These techniques can be extended to determine whether the integrity of the entire system has been compromised. While the idea sounds easy, given the power of the adversary (malware), a very careful design has to be done to prevent the malware from declaring to the server that the system is safe.

The methods presented in this work have been implemented completely in software. We opine that such approaches, judiciously combined with traditional malware prevention methods, can make computing safer without adding much overhead to the applications and operating systems. In the remainder of the paper, we present work related to our approaches and provide brief overview and implementation details of our approaches.

II. RELATED WORK

There are several variants of the buffer overflow attacks like stack overflows, heap corruption, format string attacks, integer overflow and so on [3]. C and C++ are very commonly used to develop applications; due to the efficient "unmanaged" executions these languages are not safe. A vast majority of vulnerabilities occur in programs developed with these languages [4]. Randomization is a technique to inject diversity into computer software. The first known randomization of stack frame was proposed by placing a pad of random bytes between return address and local buffers [5]. Random pads make it difficult to predict the distance between buffers and the return address on the stack. An attacker has to launch custom attacks for every copy of the randomized binary. Other than simple ASLR [2] all randomization techniques involve re-compiling the program and hence do not work for binary distributions. We implement techniques to randomize the stack and the heap frames, at a fine grain level, of every application

This material is based upon work supported in part by the National Science Foundation under Grant No. CNS-1011931. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

978-0-7695-4211-9/10 $26.00 © 2010 IEEE
DOI 10.1109/SocialCom.2010.147

in the system (without access to source code). This in turn serves to randomize relative addresses within every copy of the application, making construction of overflow based attacks difficult.

Many approaches have been developed to ensure secure software based key management. Centralized key storage employs techniques where the generation, storage, distribution, revocation and management throughout the lifetime of the key happens on a single central machine. This offers advantages such as ease of backup, recovery, and securing a single machine secures the keys [6]. Secondary storage or a detachable device has also been used to hide keys by encrypting the key with a very strong password, but this can be attacked using a key logger that logs typed passwords on the system [7]. In the same research another method is presented to store keys in the system, it involves breaking the key into multiple parts and distributing it in different places. Distributing the key reduces the memory entropy footprint making it harder to detect the pieces that comprise the key. Another solution for key management is distributed key storage using secret sharing [8]. This could be an option for large organizations, but it is not feasible for normal end users of cryptography. The solutions proposed in this paper try to prevent key theft from memory disclosure attacks, irrespective of the number of copies of the key present in memory, and prevents key exposure altogether.

Integrity measurement involves checking if the program code executing within a process, or multiple processes, is legitimate or has been tampered. It has been implemented using hardware, virtual machine monitors, and software based detection schemes. Some hardware based schemes operate off the TPM chip provided by the Trusted Computing Group [9, 10, and 11]. Some hardware based schemes operate off a co-processor which can be placed into the PCI slot of the platform [12 and 13]. Terra uses a trusted virtual machine monitor (TVMM) and partitions the hardware platform into multiple virtual machines that are isolated from one another [14].

In Pioneer [15] the integrity measurement is done without the help of hardware modules or a VMM. The verification code for the application resides on the client machine. The verifier (server) sends a random number (nonce) as a challenge to the client machine. The response to the challenge determines if the verification code has been tampered or not. The verification code then performs attestation on some entity within the machine and transfers control to it. Pioneer assumes that the challenge cannot be re-directed to another machine on a network; however in many real world scenarios a malicious program can redirect challenges to another machine which has a clean copy of the attestation code. In its checksum procedure, Pioneer incorporates the values of Program Counter and Data Pointer, both of which hold virtual memory addresses. An adversary can load another copy of the client code to be executed in a sandbox like environment and provide it the challenge. This way an adversary can obtain results of the computation that the challenge produces and return it to the verifier. In this paper remote attestation is implemented by downloading new (randomized, obfuscated, binary) attestation code for every instance of the operation. This makes it difficult for the attacker to fake any results that are produced by the attestation code.

III. Randomization of Stack Frames and Allocated Heap Chunks

If the memory layout in each copy of the binary was different on every machine it would make it extremely hard attacks to occur. The relative addresses of memory objects remain constant for every application, this allows an attacker to determine the number of bytes of offset from the current location of memory and launch attacks. The stack frame and heap chunks are randomized in this work so that every allocation of stack or heap has a different amount of "gap" added to the allocation. We do not assume existence of source code. The stack frame is randomized post-compilation by rewriting the memory accesses in binary to relocate the locations of the stack variables. The heap chunks are randomized by changing the system library code and the kernel code.

A. *Randomization of Stack Frame*

We randomize the size of the run-time stack frames to make every instance of an executing binary have a unique memory layout. The binaries are instrumented by analyzing the disassembly of the code segment in a binary. We do not inject additional bytes of code in the binary but rewrite existing bytes in the code segment. The randomization process is carried out after the application binary is installed at the end-user machine. During randomization only those instructions that are relevant to the run-time stack need to be rewritten. By shifting the vulnerable character buffer down by a random amount, the distance between the return address and the buffer becomes different for every copy of the binary. This makes it difficult to use the same attack string against different copies of the binary. Instructions of the following type need to be rewritten in the binary when adding a random pad:

- Instructions that create space on the stack frame for local variables and buffers.
- Instructions that de-allocate space used by the locals of the function on the stack frame. These instructions are executed right before the function returns. In most functions, the stack de-allocation is done implicitly by the leave instruction that restores the stack pointer to the frame pointer; hence we don't need to explicitly modify any instruction for correct de-allocation of the random pad memory.
- Instructions that access local variables and buffers on the stack frame. All local variables and buffers are accessed with the help of the frame pointer EBP. All stack locals are located below the frame buffer at lower addresses in the Intel x86 architecture. Because of the random pad, the local buffers shift further down from the frame pointer.

Implementation: The prototype randomizer was developed in C and compiled using gcc, the objdump disassembler was used for disassembly of binaries. Fig. 1 shows the flow of the randomizer. The binary file is fed to the disassembler, its

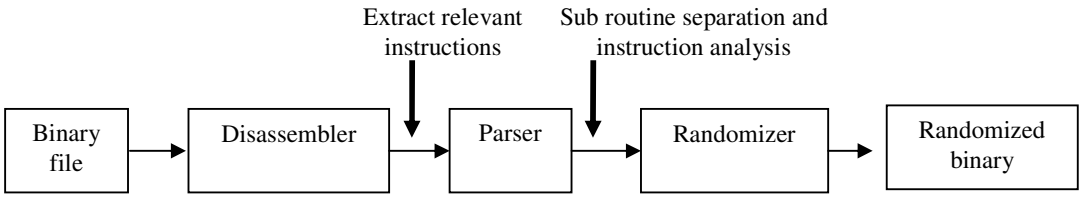

Figure 1. Workflow of randomizer

output is parsed for identification of instruction operands that need to be modified in the binary. The parser separates out and analyzes each sub-routine in order to accomplish fine-grained randomization such that every function is padded separately with a random padding conforming to the constraints of that specific function. Thereafter these instructions are directly rewritten in the binary to change the layout of the stack frames at run-time. The prototype works as a 2-pass randomizer. In the first pass, each sub-routine is analyzed to determine the maximum padding that can be provided to that routine. Every instruction in the routine that accesses memory regions has an upper limit on the relative address that can be accessed by it. We process every instruction and check the maximum available random pad to that instruction. The least of these values becomes the pad for the function. The randomizer also looks for instructions that are sensitive to alignment of memory operands and takes a conservative approach of not randomizing sub-routine with very sensitive instructions. The random pad is then clipped to the nearest factor of 32 to resolve many alignment requirements of several instructions. In certain cases it is also necessary to place an upper-limit on the maximum padding given to each sub-routine as it increases the chances of a stack overflow causing the process to crash. In the second pass, the randomizer goes through the instructions in the disassembly and locates them in the executable binary file. While tracing every instruction the randomizer also keeps track of the sub-routine in which the instruction is present. With the help of the data structure built for every sub-routine during the first pass, the randomizer statically rewrites and instruments the corresponding instruction in the binary executable.

We randomized copies of the following applications: Open Office, pidgin, pico, ls, gcc, netstat, ifconfig, route, xcalc, tail, date, nslookup, sum, head, wc, md5sum, pwd, users, cat, cksum, hostid, logname, echo, size, firefox, viewres, xwininfo, oclock, ipcs, pdfinfo, pdftotext, eject, lsmod, clear, vlc, and gij. Thus we cover both "console" applications and graphics applications. It is clear that we have a proof of concept implementation that can cover all applications that we tested it on. All the binaries used in testing were release quality, optimized utilities that are part of Linux distributions. Since we only manipulated the size of the run-time stack, we do not expect this approach to have any run-time performance penalty whatsoever. We found that on an average the randomizer modified the run-time stack of more than 75% of the sub-routines in every application. Some of the routines are not randomized as we take a conservative approach to not make any changes to routines containing alignment sensitive instructions such as FXSAVE. We are also restricted by the length of the stack allocation instruction as we do not inject additional bytes into the program. If the width of the operand on the stack allocating instruction is only one byte, we can allocate a maximum of 128 bytes of stack with such an instruction. If the routine already allocates 128 bytes of stack, then its stack frame cannot be randomized.

B. *Randomization of allocated heap chunks*

We single out the library functions that play a vital role in heap memory management (the functions that perform the free, allocate and resize operations). These functions are wrapped with randomization code, the library entry points of these functions are hooked to point to the wrappers. Source code access of binaries is not utilized as the underlying heap memory management mechanism is modified. We adopt a dual random padding strategy per every memory allocation. This is done by appending a random pad below as well as above the pointer to the heap memory chunk returned by the allocation algorithm. Fig. 2 gives an overview of this.

Implementation: This approach was implemented by identifying the memory management functions to be patched in the GNU C library. The most important of these functions that we identified are – malloc(), free(), realloc() and memalign(). Other related functions which we identified are calloc(), valloc() and pvalloc() which need not be patched as they are based entirely on malloc() in the current version of the GNU C library. We generate two random integers i and j, which are multiples of 8 to respect the internal memory alignment rules. The upper limit of the random numbers generated can be selected heuristically. These two random integers are added to the size parameter contained in the original request, making an allocation call for i+ j+ original request. A successful malloc() operation returns the pointer to a newly allocated memory chunk. The value of the pointer returned to the calling function (user application) is shifted by i bytes. These two random numbers are stored so that other memory management functions like free() and realloc() know the size of the allocated unit and calculate the actual start address of the memory chunk and thus the boundary tag information stored above it. Once a call to free() is made by the requestor function, execution enters the free() public wrapper. We extract the value i set by malloc() which lies just above the chunk address passed as an argument to free(). Using this value, the original starting address of the memory chunk's user space can be calculated. This value can be passed to the internal free() function called here forth. Similar calculations are made for realloc() and memalign() library calls.

IV. SECURE KEY STORAGE USING VMM AND DISK STRIPING

Stealing secrets from memory of executing programs is an effective method for circumventing security systems,

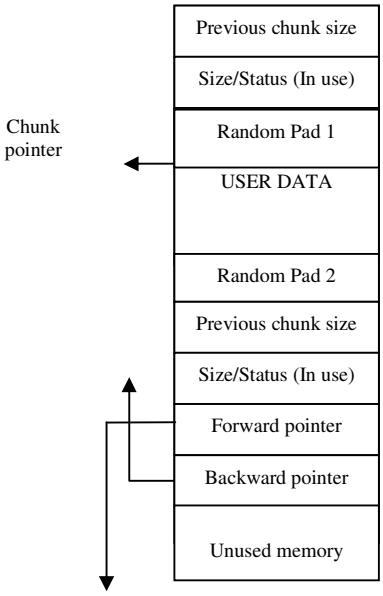

Figure 2. Allocated memory chunks in the heap

especially encryption. Encryption keys have to be stored as clear-text in memory when the application performing encryption executes. This information is susceptible to memory forensics based attacks. For example, the AACS encryption for high-definition DVD players uses a master key to keep other keys, and uses the unbroken AES encryption method. It has been documented that a particular HD-DVD encryption key was stolen from memory [16]. In this paper we present two methods of safely storing encryption keys. This first involves hypervisor support. The keys are placed in a hypervisor below the operating system. The second method presented in this paper is disk striping, in which the keys are kept on disk every time the key is not actively in use (even if the key handling application is running). The keys are split into tiny chunks of a few bits each and placed in hidden blocks of the disk that are not part of the file system.

A. *Hypervisor Based Key Storage*

Virtual Machines (VM) are primarily used for executing multiple machines on a single hardware platform. Virtual Machine Monitors (VMM) are also widely used to provide a trusted computing base to security sensitive applications as they can provide logical isolation between different VMs. We offload the cryptographic operations of the system to the secure VMM. The guest operating system(s) interact with the user and receive the request to perform cryptographic operations. The guest operating systems make a *hypercall* to the VMM to perform the actual encryption/decryption. We leverage the secure nature of the VMM where a guest OS cannot read the contents of the host VMM, but the VMM can read the contents of every guest OS. Any attacks launched by the attacker Mallory are restricted on the guest space. Due to this Mallory cannot get information about the key using forensic analysis on the guest memory. We also implement attestation which ensures that the application calling the cryptographic hypercall is legitimate.

Implementation: The system was implemented on the Linux 2.6.23 kernel. We used the same guest and host operating systems. We utilized the *lguest* modules to implement the VMM. We used the DES module to perform cryptographic routines. We added all the required hypercalls for performing these operations. We performed attestation by creating a nop placeholder for the attestation code inside the guest kernel. When the guest kernel issues an attestation request, the VMM provides executable code which is to be injected at this location. The guest kernel uses *copy_to_user*() call to inject bytes at the specified placeholder. The sequence of operations to perform cryptographic operations can be summed up as:

- The guest user space application issues requests for performing cryptographic operations. It passes the type of operation and the data to be operated upon as input. The request issues a software interrupt and the context switches from guest user space to guest kernel space.
- The guest kernel forwards the request to the trusted VMM. The secure VMM injects code into the running guest kernel on receiving a request. The injected code is responsible for returning the guest physical address where the user space program is loaded. It also brings the user application's pages into memory in case they are swapped out. The injected code is changed every time an attestation request is made.
- Control is transferred back to the VMM which reads the contents of the user space program directly from the memory, using the address obtained above. The secure VMM computes and compares the hash value of the memory content to pre computed hash values obtained from the original binary image of the program.
- The requested cryptographic operations are performed by the VMM, and the results written back to the memory location passed by the guest kernel, the guest kernel copies the results to the guest user space.

B. *Disk StripingBased Storage*

We hide the key on secondary storage by writing to the unused sectors of files on the hard disk without using the file system calls of the OS. Each file on the storage media has an end of file (EOF) marker. The OS allocates space for files in disk blocks and does not reclaim the space beyond the EOF marker if a particular block is partially used. The space beyond the EOF on a sector is used in this research to store the key. We do not add key information to random files, but we place new files during installation of the code. Once the storage area is determined, we scatter the key throughout this area such that the attacker cannot retrieve the key even after knowing the sector where it is stored. We refer to the location of a bit of the key in scattered array as bit-address. Every bit of the key has a bit-address associated with it which forms the bit-address pattern. This bit-address pattern is unique to every installation

Position of Fetch blocks	Location of Jump to fetch blocks	Total size of all fetch blocks
0x21B7	0xCF7	0xA40
0x21DA	0xCC5	0xA03
0x21B1	0xCF2	0xA86
0x21C0	0xCBB	0xA1F

TABLE I. DATA FROM DIFFERENT INSTALLATION INSTANCES

of the system. If the bit-address pattern is stored directly, then it can be easily read by the attacker. Instead of storing the bit-address pattern, the bit-address fetching block generates the address of each bit at run time. There is one logical block of code per bit address to be generated. The bit-address fetching block is generated for the application during installation time. We generate a bit-address and then generate the bit-address fetching block to match the addresses. This is achieved by performing random arithmetic operations on the result value till we achieve the desired value for that particular bit. The arithmetic operations are chosen from a pool of operations. We also obfuscate the location of the bit-address fetching block in the binary.

Implementation: This part of the research was implemented using the gcc compiler, and Ubuntu 8.04 Linux OS. We were able to randomize the locations of the bit-address fetch blocks and the locations to the jump call to all the blocks. Table 1 shows a small sample of data from the multiple installation sequences. It can be seen that the size of each bit-address fetching block was also different in each installation.

V. INTEGRITY MEASUREMENT USING REMOTE ATTESTATION

Remote attestation is a framework for allowing a remote entity to obtain integrity measurements on an un-trusted client machine. In order for remote attestation to work, we need to have access to a remote verifier (Trent) that is a trusted (uncompromised) host and is accessible over a network. A single trusted host can be used for a large number of clients; ensuring that sensitive applications running on the clients are not tampered by malicious code on the client machines. In this framework we determine whether an application ($\mathcal{P}$) installed by Alice (the client) has been modified. In the consumer computing scenario, we envision the deployment of "attestation servers", where an end-user contracts with the service provider to test the safety of the applications on the end-platform. This is similar to the way virus detector updates are done today.

Remote Attestation has been implemented traditionally with the help of hardware modules [9, 10, and 11] and the use of VMM [17] has also been suggested. It involves the trusted server (Trent) communicating with the hardware device installed on the client's (Alice) machine. However, these modules are unsuitable for legacy platforms, and have the stigma of Digital Rights Management attached with them. The use of a VMM also requires greater hardware resources and compute power. In our framework remote attestation is implemented entirely in software without kernel support. Operating system support is not used in this framework as it would require a secure OS, or a loadable kernel module that performs the attestation. The first scenario is unlikely to occur, and the second scenario would require frequent human interaction to load the kernel modules in the system (to prevent automatic exploits of the kernel loading modules). The approach taken in this paper is designed to detect changes made to the code section of a process. Trent is a trusted entity who has knowledge of what the structure of an un-tampered copy of the process ($\mathcal{P}$) to be verified. Trent provides executable code ($\mathcal{C}$) to Alice which Alice injects on $\mathcal{P}$. $\mathcal{C}$ takes overlapping MD5 hashes on the sub-regions of $\mathcal{P}$ and returns the results to Trent.

A software protocol provides opportunities for an attacker (Mallory) to forge results. The attacker (Mallory) can perform a replay attack in which Trent is provided with results that are the response to a previous attestation challenge. Mallory may *tamper* with the results generated by the attestation code to provide the expected results to Trent. Mallory may *re-direct* the challenge to another machine which executes a clean copy of the application $\mathcal{P}$, or Mallory may execute the challenge inside a sandbox to determine its results and send the results to Trent. This paper addresses these by incorporating specialized tests and generates random code to mitigate the effects of these attacks. We obtain machine identifiers through system interrupts to determine whether the challenge was replayed. We take measurements on the client platform that determine whether the attestation code was executed in a sandbox. Lastly, we perform extensive randomization of the attestation code by changing the arithmetic operations and memory locations read by every instruction.

When $\mathcal{P}$ contacts Trent the remote attestation starts. Trent provides $\mathcal{P}$ with a binary attester code $\mathcal{C}$ (that is signed by Trent). $\mathcal{C}$ is generated each time it is "provided" and is composed of randomized and obfuscated binary, and hence it is difficult to reverse engineer $\mathcal{C}$. Since $\mathcal{C}$ is downloaded code, Trent has to be trusted not to provide malware to Alice. $\mathcal{P}$ runs $\mathcal{C}$, and $\mathcal{C}$ hashes the memory space of $\mathcal{P}$ in random overlapping sections and then encrypts the hashes with a nonce that is contained in $\mathcal{C}$ and sent to Trent. The nonce is located in a different location for each instance of $\mathcal{C}$ making it impossible for a compromised $\mathcal{P}$ to mimic $\mathcal{C}$'s behavior. When Trent gets the results from $\mathcal{C}$ it verifies that $\mathcal{P}$ has not been tampered with and it is executing correctly. Now that we know $\mathcal{P}$ is correct, $\mathcal{P}$ can be entrusted to verify the integrity of the security sensitive application that execute on Alice. Fig. 3 shows an overview of the framework.

Implementation: The remote attestation scheme was implemented on Ubuntu 8.04 (Linux 32 bit) operating system using the gcc compiler; the application $\mathcal{P}$ and attestation code $\mathcal{C}$ were written in the C language. $\mathcal{C}$ computes a MD5 hash of $\mathcal{P}$ to determine if the code section has been tampered. Downloading MD5 code is an expensive operation as it is large, also MD5 code cannot be randomized as it may lose its properties, and hence the MD5 code permanently resides on $\mathcal{P}$. To prevent Mallory from exploiting this aspect a two phase hash protocol is implemented. Trent places a mathematical checksum inside $\mathcal{C}$ which computes the checksum on the

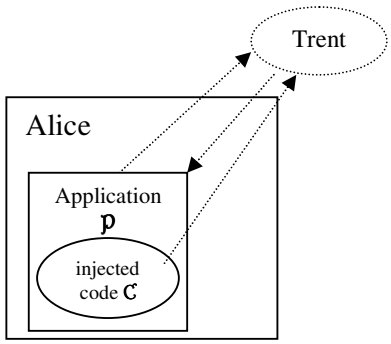

Figure 3. Remote attestation

region of p containing the MD5 executable code along with some other selected regions. The operations of the mathematical checksum are randomized by creating a pool of operations for every instruction, and selecting one instruction randomly from each pool. Trent receives the results of the arithmetic checksum, verifies the results, and sends a message back to C which proceeds with the rest of the protocol if Trent responds in affirmative. The checksums are taken on overlapping sub regions to make prediction of results more difficult for Mallory. This creates multiple levels of indeterminacy for an attack to take place. To determine whether C was bounced to another machine, Trent obtains the address of the machine that C is executing on. Trent had received an attestation request from Alice, hence has access to the IP address of M_{Alice}. If C returns the IP address of the machine it is executing on, Trent can determine if both values are the same. Although IP addresses are dynamic, there is little possibility that any machine will change its IP address in the small time window between a request by Alice to measurements being taken and provided to Trent. C determines the IP address of M_{Alice} using system interrupts. An attacker will find it difficult to tamper with interrupts on a system given that there are many interrupts and changing their implementation is not simple. To determine that p was not executed in a sandbox environment, C determines the number of processes having an open connection to Trent on the client machine. This is obtained by determining the remote address and remote port combinations on each of the port descriptors in the system. C performs communication to Trent by using the socket descriptor provided by p. This implies that in a pristine situation there must be only one such descriptor on the entire system, and the process utilizing it must be the process inside which C is executing. If there is only one such process, C computes its own process id and compares the two values. An error is reported if these conditions are not met.

VI. CONCLUSION

In this paper we presented solutions to detect the presence of compromised binaries, storing secret keys in memory, and modifying the memory layout of binaries entirely in software. These three scenarios represent most of the attacks that occur in the computing world. Mitigating these situations allows us to improve the security of end user platforms. These techniques can be used independently and are *not dependant on the software vendors to provide source code of their products.*

Future work would be to extend the techniques described above to enhance the security of the operating system itself by randomizing the OS and/or providing remote attestation of the OS code. Such attempts would of course raisse other issues such as a variance in the OS code due to hardware differences amongst makes and models of computers. Work is needed to study the techniques of combining the approaches into a single system with multiple defense mechanisms. Randomization of the address space does imply changes to the code itself and hence code on different machines may have different MD-5 hashes and these have to be traced by the remote attester.

REFERENCES

[1] R. Srinivasan and P. Dasgupta, “Towards more effective virus detectors,” Communications of the Computer Society of India, vol 31-5, pp. 21-23. August 2007.

[2] Web link, ASLR: “Address space layout randomization,” retrieved on April 25 2010, http://pax.grsecurity.net/docs/aslr.txt

[3] J. Foster, V. Osipov, N. Bhalla, and N. Heinen, “Buffer Overflow Attacks: Detect, Exploit, Prevent,” Syngress Publishing: 2005.

[4] R. Seacord, “Secure Coding in C and C++,” Addison-Wesley: 2005.

[5] S. Forrest, A. Somayaji, and D.H. Ackley, “Building diverse computer systems,” HOTOS ’97: Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI), IEEE Computer Society, pages 67–72, May 1997.

[6] Web Link, "nCipher Solutions," Retrieved on April 20 2010, http://iss.thalesgroup.com/Resources/Product%20Data%20Sheets/keyAuthority.aspx

[7] A. Shamir and N. van Someren, “Playing ‘Hide and Seek’ with Stored Keys,” Third International Conference on Financial Cryptography, pp. 118-124, 1999.

[8] R. Canetti, Y. Dodis, S. Halevi, E. Kushilevitz and A. Sahai, “Exposure-resilient functions and all-or-nothing transforms,” Advances in Cryptology – EUROCRYPT, pp. 453-469, 2000.

[9] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn, “Design and implementation of a TCG-based integrity measurement architecture,” Proceedings of the 13th USENIX Security Symposium, pp. 223 -228, 2004

[10] K. Goldman, R. Perez, and R. Sailer, “Linking remote attestation to secure tunnel endpoints,” STC '06: Proceedings of the first ACM workshop on Scalable trusted computing, pp. 21 - 24, 2006.

[11] F. Stumpf, O. Tafreschi, P. Röder, and C. Eckert, “A robust integrity reporting protocol for remote attestation,” WATC’06: Second Workshop on Advances in Trusted Computing, 2006.

[12] L. Wang and P. Dasgupta, “Kernel and application integrity assurance: Ensuring freedom from rootkits and malware in a computer system,” Advanced Information Networking and Applications Workshops, pp. 583 – 589, 2007

[13] N.L. Petroni Jr., T. Fraser, J. Molina, and W.A. Arbaugh, “Copilot-a coprocessor-based kernel runtime integrity monitor,” Proceedings of the 13th conference on USENIX Security Symposium, vol 13, 2004.

[14] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, D. Boneh, “Terra: A virtual machine-based platform for trusted computing,” ACM SIGOPS Operating Systems Review, pp. 193 – 206, 2003

[15] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. Van Doorn, and P. Khosla , “Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems,” ACM SIGOPS Operating Systems Review, vol 39 -5, pp. 1 – 16, 2005

[16] Web link, “HD-DVD Content Protection already hacked?” Retrieved on April 4 2010, http://www.techamok.com/?pid=1849

[17] R. Sahita, U. Savagaonkar, P. Dewan, and D. Durham, “Mitigating the Lying-Endpoint Problem in Virtualized Network Access Frameworks,” Springer: Managing Virtualization of Networks and Services, pp. 135 – 146, 2007.