Spring 2011

Dynamic Code Checksum Generator

Ashish Sharma
San Jose State University

Dynamic Code Checksum Generator

A Research Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

By:

Ashish Sharma

Spring 2011

SAN JOSÉ STATE UNIVERSITY

The Undersigned Thesis Committee Approves the Project-Thesis Titled

Dynamic Code Checksum Generator

By

Ashish Sharma

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Mark Stamp,	Department of Computer Science	Date
Dr. Robert Chun,	Department of Computer Science	Date
Mr. Nitin Jagga,	Cisco Systems	Date

APPROVED FOR THE UNIVERSITY

Associated Dean	Office of Graduate Studies and Research	Date

Abstract

A checksum (i.e., a cryptographic hash) of a file can be used as an integrity check, if an attacker tries to change the code in an executable file, a checksum can be used to detect the tampering. While it is easy to compute a checksum for any static file, it is possible for an attacker to tamper with an executable file as it is being loaded into memory, or after it has been loaded. Therefore, it would be more useful to checksum an executable file dynamically only after the file has been loaded into memory. However, checksumming dynamic code is much more challenging than dealing with static code – the code can be loaded into different locations in memory, and parts of the code will change depending on where the code resides in memory (addresses, labels, etc.).

Windows Vista and later versions of Windows include a new technology known as Address Space Layout Randomization (ASLR). ASLR, which serves as a defense against buffer overflow attacks, causes the executable file to be loaded at a randomly-selected location in memory. The goal of this project is to develop a robust and efficient technique for computing the cryptographic hash of a dynamic executable in the presence of ASLR.

Acknowledgements

I would like to thank Dr. Mark Stamp for showing faith in me, and for giving me this remarkable opportunity to work under his guidance for this master's degree project. I also wish to thank him for his patience, suggestions, ideas, and for the inspiration that he has given to me, without which this project would not have been possible.

I would also like to thank Prof. Barbara Wesley for guiding me with the writing of this project.

Contents

1. Introduction

With the increase in the use of computers, there has been a tremendous increase in the number of desktop applications as well. Because many of these new desktop applications are no longer free, there has also been increase in the breaking of these applications. The so-called “bad guys” in the world of security pursue malicious activities that disrupt the normal flow of productivity.

In 1998 – when virus development was in its early stages – a wide variety of new infection techniques were introduced. Most of these viruses took advantage of PE files, attacking them by adding extra sections to the file or by adding the malicious code to the empty spaces between sections. A PE file is file format for executable or .exe files [6]. An example of this is the Virus.Win32.IKX virus, which would look for gaps in the virtual image of a file and add code in the middle and in the last section. The virus would then change the entry point and fix section headers [15]. Some viruses would add extra sections – like the .text section, where the programmable code is present – and then change the entry point to the newly-added section. Another example of such a virus is Win95.invir.7051 [16], which would infect those PE files that are opened, renamed or the file attributes are read or set. To infect a file, the virus would encrypt the code and add code to the end of the last section.

One common trait among these viruses is that they add extra malicious code to the already-present code – the attacker would infect the PE file with additional malicious code and use that file to attack the system. However, this infection technique was easy to discover, as there would be suspicious content in each of these virus-infected .exe’s.

Hence, virus writers introduced newer techniques to infect files, such as encrypting the suspicious content. Every time application developers added new security features, virus writers would develop a work around for them. One way to increase the security of an application is the *checksum*, or *hash validation* the checksum or hash of the file is calculated and then compared to the file. If the attacker tries to change the file by reverse-engineering it and adding extra sections, the checksum of the file changes and the validation fails. The code can take appropriate action if it encounters any such abnormality. Another use of checksum is to protect files from modification, for which a checksum of the files is calculated and saved securely. Operating systems use this technique for security. After a regular interval of time, this checksum is recalculated to confirm that these files are not modified or changed. But, this technique is not used for all files, as there are files in a system that change often [17].

A security feature added by Microsoft in their latest operating systems is *Address Space Layout Randomization* (ASLR). ASLR is not a complete security measure, but it provides an enhancement to the present system. ASLR adds a little complexity to the loader, but in return it allows more secure applications. ASLR gives us a general defense mechanism against attacks on memory corruption [18].

Windows XP and earlier operating systems did not have ASLR, so it was easy for an attacker to intrude into a system. The attacker took advantage of this vulnerability and was able to launch different types of attacks on a PE file. One of the most common attacks was the buffer overflow attack, a kind of anomaly in code that uses buffers to write the data. An executable code is made of three main memories: instruction memory, heap memory, and stack memory. Instruction memory is comprised of executable code.

Stack memory is used to store local variables, buffers, and return addresses. Heap memory is used to store dynamic length data. The stack memory is filled "first in, last out" – what goes into the stack first comes out last. Since this stack contains function arguments, local variables, and return addresses one over the other, it is very possible to overrun these memory locations if any of the variables or arguments is more than the assigned length. An attacker can take advantage of this architecture and overrun the return address with its malicious code by giving a long input value to the variables or arguments. The attacker can add its malicious code as input at the right location and can therefore gain control over the system [19]. ASLR prevents buffer overflow attacks by loading various parts of an executable on random addresses. Libraries, stack, heap, and executable code, all of them, are loaded into different memory locations [20].

ASLR has been very helpful in preventing buffer overflow attacks, but is not completely secure enough to prevent them. ASLR loads the executable in any of the memory locations from the 256 available memory spaces, like an attacker can still brute force these addresses until it succeeds. There are also other ways to determine the memory location where an exe is loaded, such as coercing an application to leak one of its addresses – for example, leaking the address of a function inside the dll [21].

As such, it is important to check the integrity of the executable after it has been loaded, since an attacker could otherwise use any of the methods mentioned above and cause harm to the system. We can check the integrity of an exe by comparing the checksum of the bytes of the programmable code when it has been loaded into memory. Since the addresses are changed every time the executable is loaded, a simple checksum or hash comparison is of no use, as each time the checksum value would be different. The main

aim of this project is to calculate the checksum of the loaded executable in such a way that its value is always constant. In order to achieve this, we skip the bytes that would be changed due to ASLR.

2. Background

There have been many attacks on exe's. Until 1999, only a-hundred-or-so Windows viruses existed, such as Win32/Cabanas, Win95/Anxiety, and Win95/Marburg [22]. These viruses were nightmares for Microsoft's executable files. Now, the numbers of such viruses has grown enormously. The most common attack is to reverse-engineer PE files and change the assembly code. Reverse-engineering is the process of extracting assembly instructions of an executable file – the attacker opens the PE file with any free disassembly and debugging tool, like Ollydbg or IDA Pro, and then makes changes to the assembly code to bypass the security [23],[24]. There is no permanent solution to stop an attacker from reverse-engineering and changing an exe's instruction; application developers can only prevent an attacker from anti-debugging. Windows uses several anti-debugging techniques, some of which are as follows:

1. IsDebuggerPresent: This is a Windows function that will return 1 if the process is being debugged and 0 otherwise.

2. CheckRemoteDebuggerPresent: This is another function provided in Win32 API. It will check if any remote process is debugged.

3. NtQueryInformationProcess: This function will retrieve information about any specified process.

4. NtGlobalFlag: This is a dword value inside the process environment block. It has certain flags which if set indicate that a process is being debugged [25].

Security experts can make the work of an attacker difficult, but not impossible, by adding opaque predicates, lots of junk code to confuse the attacker or by encrypting the code, or by using other defensive techniques. Opaque predicates are conditions or expressions that always return a specific result, generally conditional statements that always return true or false. The actual code will ignore the dead code, but the disassembler would not be able to ignore the junk code inside the opaque predicate [4]. These techniques make an application more secure and less susceptible to attacks.

Checksum validation is another technique to increase the security of an application by detecting undesirable changes in the program. Checksum is like a unique fingerprint of a file. To compute a checksum, all the bytes are selected and then a checksum algorithm is applied to those bytes to compute a unique value, which cannot regenerate the original input from the given output. If there is even a single change in the byte, it would return a different checksum. Then, this checksum (or hash value) of a PE file (or exe file) is calculated and compared. If the attacker tries to make any malicious change in the code, the hash value of the PE file would also change and hence the validation will fail as well.

2.1 Attacks on PE files

Attackers have targeted exe files for a long time. In the late 90's, the count of Win32 viruses was relatively small – somewhere in the hundreds – but the growth has been

exponential ever since [7]. Virus writers have created different versions of viruses and also different ways to keep them stealthy, but harmful. The most common types of attacks on an exe file are as follows:

1. Appending an extra section: This is the most common technique used by virus writers to infect an exe and get into the system. The attacker attaches an extra section to the last section of an exe file, so that when the exe is run in a system it also runs the malicious code attached in the last section.

2. Changing the entry point: Here, the attacker changes the entry point of an exe to the virus code. This way, when an exe is executed it will take the pointer to the location where virus code is present.

3. Disassembling and patching: Another kind of attack to breach a system's security is to disassemble the exe into assembly code. The attacker can then change the assembly code, for example changing instructions for where the password is verified and then patching it so that the exe can be used without any login or key [7].

In general, a virus attaches itself to a PE file or by adding a new section to the PE file, and when these executable files are run, it also runs the malicious code. Mostly these viruses infect system executables, as they are running all the time when the system is running – this makes such viruses stealthier. These viruses are also called "cavity viruses" [17]. Some examples of these include,

1. CIH, also known as Chernobyl or Spacefiller [8]: This virus first appeared in 1998, targeting Microsoft Windows. One of its infected targets was a demo game by

Activision called SiN. This virus was only1KB large, and would fill the gaps present inside the PE file and then reassemble itself via a small function.

2. 5lo Virus [9]: This is also one of the viruses that infect exe files. First discovered in 1992, it would append itself at the end of an exe file and install itself when the exe file is running.

3. PE File Format

A *Portable Executable file* (PE File) is an executable file that is used to install any application on the system [6]. A PE file when on disk is an ordinary file, but when loaded into memory it becomes a module, and the file is mapped to that module [2]. A PE file is made of various sections and headers, each of which has a specific purpose. The memory location where this PE file is going to be loaded is defined under these section headers.

Before taking any action on a PE file, the loader parses all the headers and then proceeds to load the file. It also determines how much space should be allotted in the memory. All of this information is stored inside the PE headers. One important thing to note is that a PE file is not mapped to memory as single memory mapped file. Instead, The Windows loader looks into the PE file to decide which portions of file should be mapped. The offset of a file is different from the loaded process. The figure below provides details of this memory mapping.

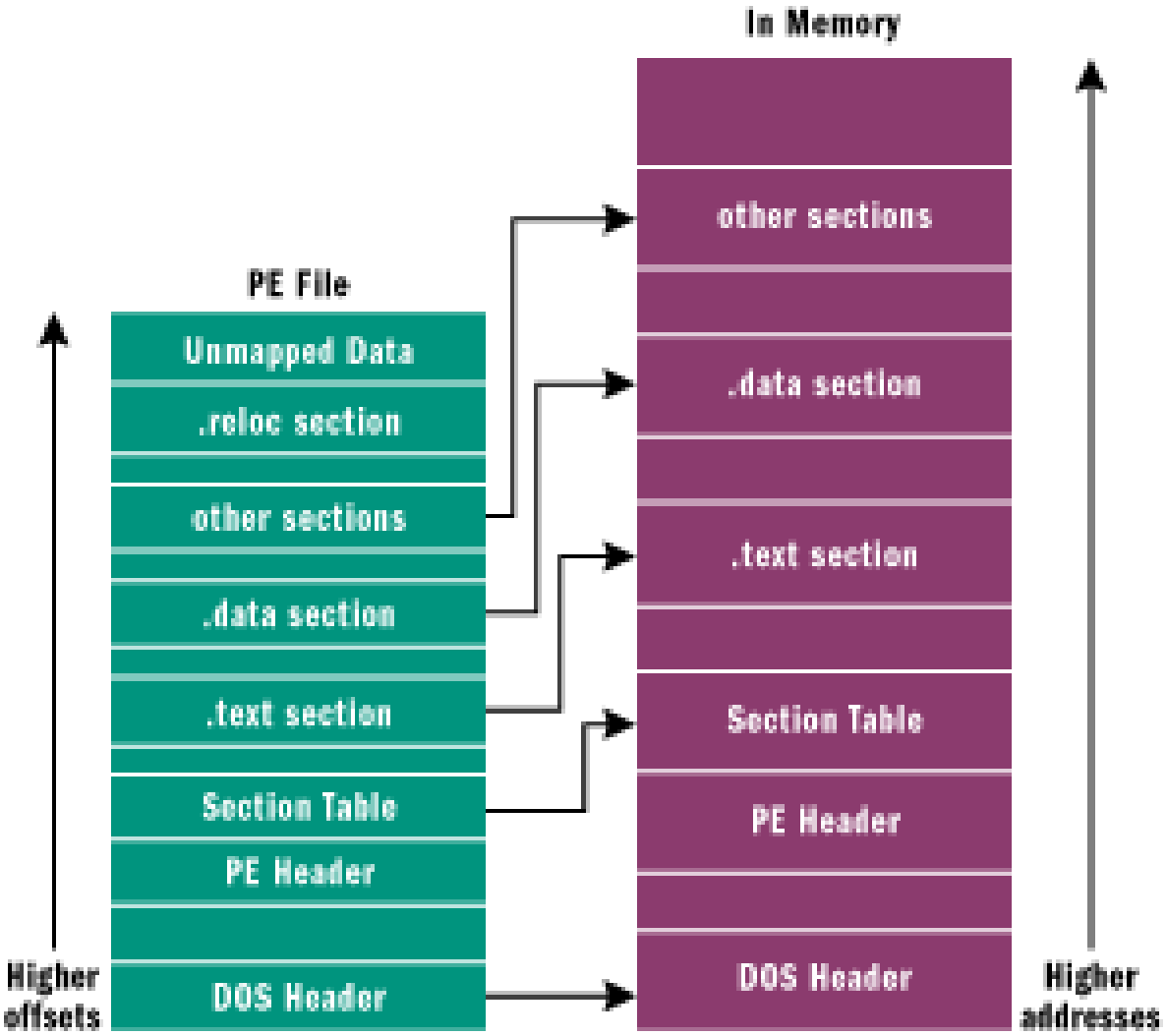

Figure 1: PE file memory mapping. [13]

Below is a diagram of a PE file. It shows the PE file headers.

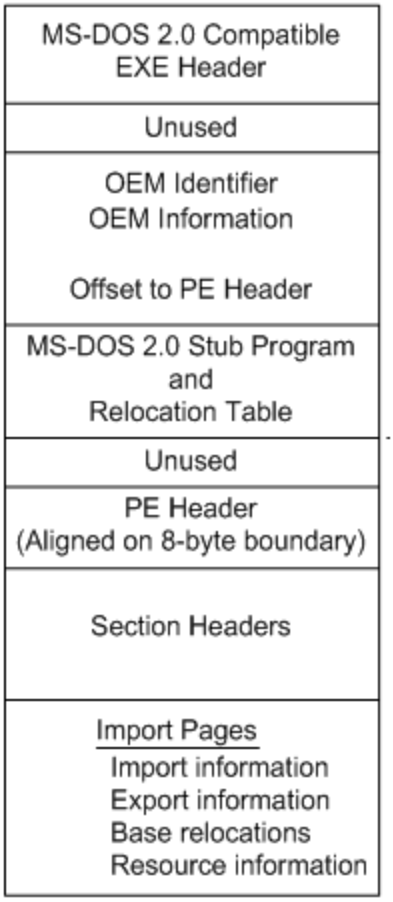

Figure 2: PE file layout [2]

3.1 File Headers

A PE file header stores information about the file, such as image size, stack size, and a number of various sections. It also gives information such as whether it is an exe file or a dll. The PE file header is made up of MS-DOS stub, PE Signature, COFF File header, and an Optional Header.

3.1.1 MS-DOS Stub

MS-DOS stub determines that the image file is a valid file and can be run under MS-DOS [1]. The linker triggers the message “This program cannot be run in DOS mode” if the image file is not meant for DOS mode [2].

3.1.2 Signature

After the MS-DOS stub is a 4-byte signature of the file. This identifies the file as a PE-format image file [2].

3.1.3 Optional Header

Every image file has this header, and it provides information to the loader. It’s called “optional” because some files do not have it, specifically object files. It provides information to the loader that includes the following:

1. Magic number: This field determines whether this PE file is for a 32-bit address space or a 64-bit address space.

2. Standard Fields: These are the first eight fields of the optional header. These eight fields contain general information about the image file. These fields are useful for loading and running a PE file. These fields include SizeOfCode, AddressOfEntryPoint, SizeOfInitializedData, SizeOfUninitializedData, BaseOfCode, and MagicNumber.

3. Windows Specific Field: The next 21 fields contain additional information for the linker and loader for executables that are running on the Windows platform. These

fields also contain information about Windows subsystems, determining which subsystem is required by the image file to run. For example, the image file may require a dll file, a device driver, or native windows processes [2].

3.2 Section Headers

This table is right after the Optional Header. It basically stores information about each section in the image file [2]. These sections are sorted by their starting addresses, also called *Relative Virtual Addresses*. The PE file has .text and .data sections stored in different segments in the file. For each section, there is a section header or an array structure. These array structures store the information of each section separately, without combining them. Each section header reserves 40 bytes per entry, and has information such as,

1. Name: This is simply the name of the section header.
2. Virtual Size: This stores the total size of the section when loaded into memory.
3. Virtual Address: This is the address of the first byte of the section relative to the image base when the section is loaded into memory.
4. Size of Raw Data: This is the size of the initialized data on the disk.
5. Pointer to Raw Data: This is the file pointer to the first page of the section.
6. Pointer to Relocations: This is the file pointer to the beginning of the relocation entries for the section.

7. Pointer to Line Numbers: This is the file that points to the beginning of the line-number entries for the section.

8. Number of Relocations: This is the number of relocation entries for the section.

9. Number of Line Numbers: This is the number of line-number entries for the section.

10. Characteristics: This field describes the characteristics of the section [2].

There are also various section flags. Section flags detail the characteristics of a section. Some of the important flags include,

1. IMAGE_SCN_CNT_CODE: If this flag is set, it means that this section has executable code in it.

2. IMAGE_SCN_CNT_INITIALIZED_DATA: If this flag is set, it means that this section has data that is initialized.

3. IMAGE_SCN_MEM_EXECUTE: This flag indicates that this section has code that can be executed.

4. IMAGE_SCN_MEM_READ: This flag marks a readable section.

5. IMAGE_SCN_MEM_WRITE: This flag marks a writable section [10].

3.3 Special Sections

Win32 loaders process these special sections and the content of such sections. These sections have flags with some special value, which are understood by the loaders. Some of the reserved sections include,

1. .text: This section contains executable code. One of the flags for this section is IMAGE_SCN_CNT_CODE – if the value of this flag is set, it means that this section has executable code.

2. .data: This section contains initialized data, marked with IMAGE_SCN_MEM_READ and IMAGE_SCN_MEM_WRITE – if these flags are set, then this section can be read and written to.

3. .rsrc: This is the resource directory section. It has a flag IMAGE_SCN_CNT_INITIALIZED_DATA – this flag indicates that the section has initialized data.

4. .reloc: This section gives information about image relocations. Its flag IMAGE_SCN_MEM_DISCARDABLE means that this section if needed can be discarded.

5. .xdata: This section gives information about exceptions [2].

4 Address Space Layout Randomization (ASLR)

ASLR is a security mechanism integrated into Windows Vista and later versions, the aim of which is to reduce the effectiveness of prevalent exploit attempts [3]. This mechanism prevents PE files from always loading into specific memory locations. This makes it difficult for an attacker to launch a buffer overflow attack. Buffer overflow attacks take advantage of vulnerabilities in a program, and come into play when an attacker tries to add more data to a buffer than the allocated space in the memory. An executable code stores its local variables and function return addresses onto a stack. When a variable is given a value greater than the size of the memory allocated, the variable does not discard those extra bits; instead, it overwrites the memory following it. An attacker can take advantage of this weakness and add malicious code as an input to a variable, and overwrite a return address with a different return address where evil code is present. So, when the execution pointer reaches that overwritten return address, the execution pointer then goes to the attacker's code. [19]. As mentioned earlier, to combat this ASLR loads a file into random addresses. This makes it difficult for an attacker to determine where an exe would be loaded, so that the attacker is not able to guess where to return the execution pointer.

ASLR is used not only in Windows operating systems, but has also been integrated into other operating systems, including Linux, Mac OS, and Open BSD.

4.1. How does ASLR randomize the memory?

According to Microsoft, any executable that contains PE file headers is compatible to be used for ASLR [3]. Operating systems that support ASLR use a relative virtual address, which is created by adding the base address (where the PE file is supposed to be loaded), with the virtual address (assigned by the operating system). This virtual address is assigned from a range of 256 values, which is quite a wide range from which to guess randomly. The operating system selects a random image offset, selected only once per re-boot. All images are loaded for a process on this offset. Even dll's are loaded with random offsets, but because the image offset is constant, dll's that are shared between processes are loaded at the same address for all processes. Further, the thread stack and process heap are placed randomly to different memory locations.

4.2. ASLR Analysis

Every time the system is booted, the virtual address is changed, and therefore the PE file is loaded on a different memory address. Because of this, it is almost impossible for an attacker to guess onto which location the PE file or the executable is going to be loaded. Below in Figure 3, the distribution of stack addresses shows a uniform distribution. This indicates that the stack address should not be easy to predict. However, it is nonetheless possible to make de-randomization attacks on the ASLR [28].

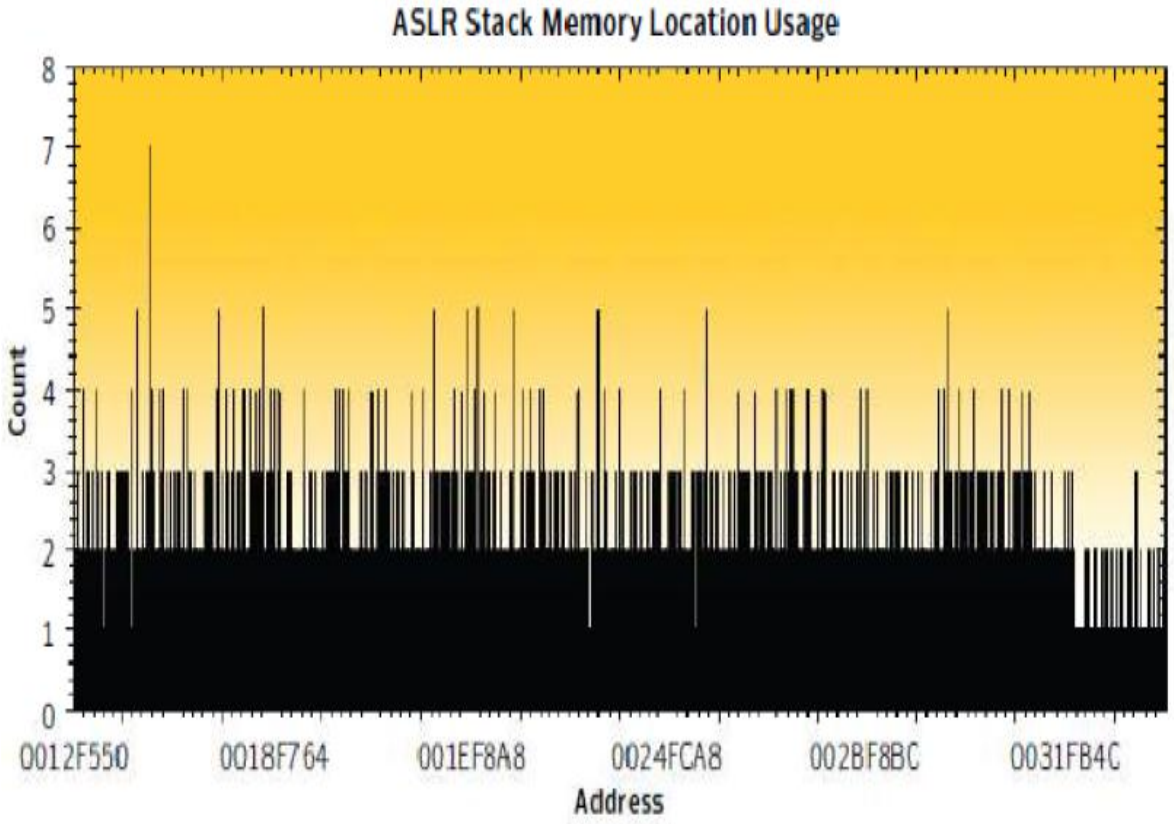

Figure 3: Distribution of Stack Addresses [3].

4.3. ASLR on Windows

Executable images that contain PE headers are eligible for ASLR. This can be done by setting the value of a bit to 0X40 in the PE Header. The option "/dynamicbase" can also be set using Microsoft Visual Studio. A random offset of the global image is selected, which changes only after a reboot. This image offset could be anywhere in the range of 256 locations. ASLR also randomizes the thread stack and process heap. Starting with the stack address, ASLR randomizes the stack address in the range of any of the 32 locations. After the stack randomization, it also randomizes the process heap in the range of any of the 32 locations. These locations are selected so as not to overwrite the already-placed

stack addresses. The starting address of the Process Environment Block (PEB), which is basically the data structure used by the OS, is also a random selection. This technique was present in earlier versions of Windows, including XP.

An experiment further demonstrates this randomness. An ASLR-compatible executable was run several times to measure the randomness. The executable was run for 11,500 times – each time the heap address of the exe was recorded, and then a graph was plotted with this data. As we can see in Figure 4, there are no noticeable patterns in the uniform distribution of HeapAlloc addresses. The x-axis of the graph is the number of times the program was run, and the y-axis represents the addresses selected by the ASLR while running the program.

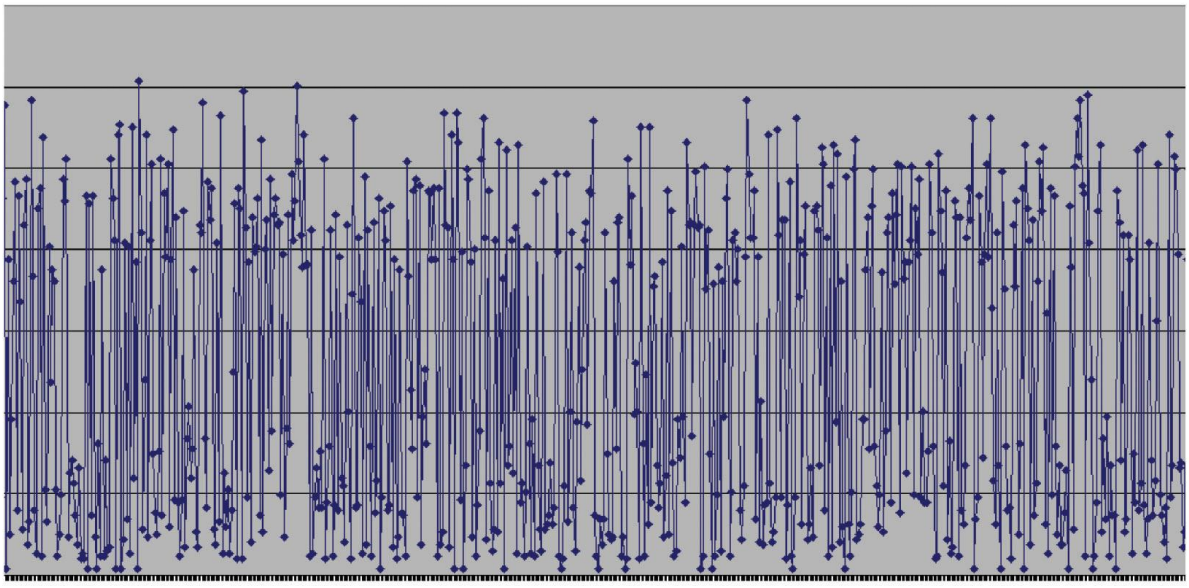

Figure 4. 11,500 HeapAlloc samples [3]

5. Cryptographic Hash Function

Cryptographic hash function is a technique by which a block of bytes is taken and after a certain number of calculations returns back a fixed size of bytes. A change in even a single bit of the input data changes the final result – i.e., the cryptographic hash value. An ideal candidate for a cryptographic hash function would be one that satisfies the following conditions [11]:

1. It should be easy to calculate the hash value of any kind of bytes given as input.

2. It should not be possible to calculate the input bytes from a given hash – i.e., it should be irreversible.

3. Every input should have a unique hash; it should be impossible to find any two different inputs with the same hash value.

4. If there is a change in even one bit of the input bytes, it should result in a different hash output value.

Such hash functions have now been used for a long time, especially in the field of Information Security. The most common use of these hash functions are,

1. Login Passwords: Almost all logins use cryptographic hashes. When a user inputs the password, the string is taken as an input and a hash value is calculated. This hash value is then compared to the stored hash value of the original password; if the hash matches, the user is authenticated. In this way, even if an attacker manages to acquire the hash dictionary or database, he would not be able to generate the password back from those hash values.

2. Digital Signatures: Digital signatures are used to verify the authenticity of a document or a digital message. If the sender sends a digital message to the receiver with the hash value of the message, the receiver can then verify the authenticity of the message by comparing the hash of the message. This is a receipt that there were no alterations made in the message, especially when the message was sent via an unreliable source such as a network where everyone can see the traffic flow [12].

3. Data Removal: A less-common use of hash function is to remove redundant data. If there is a list of multiple long data items, then instead of comparing each and every value a hash can be calculated and stored. Comparing long data items would take a lot

of time and memory space, but if the hash value is stored then we can compare these hash values relatively efficiently.

The algorithm of these cryptographic hash functions must be very accurate – even a small vulnerability could lead to an attack. There are many hash functions, but some of the famous and strong functions include MD5, SHA-1, HMAC, and RIPEMD [11].

The aim of this project is to create a tool that can determine the accurate cryptographic hash value of any PE file in the presence of ASLR. Since ASLR loads the PE file onto a new location every time it is rebooted, there is a change in the instructions every time an offset is added to the address. For example, if there is a mov instruction

mov eax, [some address]

Then due to the presence of ASLR this instruction changes to

mov eax, [some address + offset]

Not only mov instructions, but *any* instructions that have an address are changed. Because of this change in the instruction bytes the checksum is different, as these exe's will be loaded into a different memory address with a different offset. So, to calculate the checksum, we need to remove these addresses from the bytes for which we need to calculate the checksum or hash value.

We can compare this hash value just before the main execution of the application, and if there is any discrepancy in the hash value or checksum the program can exit its

execution. If an attacker breaks into the code and makes changes, this hash validation will fail and the attack will be unsuccessful.

6. Implementation

6.1. PE header Files

The PE header provides information about the exe file when it is running in the memory: the number of sections, base address, size of code, number of relocations, section which contains the code, path of the original exe, and image base (the preferred location of the exe to be loaded into the memory).

6.2. Information after loading the file

Once the PE files have been loaded into memory, our tool will fetch either its entry point or the actual loading memory address. The Windows API has specific functions, such as the Process API and Module API functions provided by Microsoft Developer Network (MSDN) [5]. These functions will help in fetching the data from the process loaded into memory.

6.3. Reading the code section

Once the loaded memory address and the size of the .text section have been inferred from the file, the Process API functions provide read access to the whole code section of the memory. This can be used to deduce all the bytes of a process or module.

6.4. Computing the hash

The hash value algorithm gathers data from the .text section. The starting address is the entry point of the algorithm, and the hash value algorithm then computes each and every byte until the last memory of the section. For this hash to be accurate, we need to remove all the bytes from the code section that have addresses in them, as these instructions are modified every time the ASLR loads the PE file into memory. This affects the hash value, as it will have different bytes each time. Since these instructions can alter the hash value, before considering the bytes each relocation address is compared to see if it lies between the specific ranges of addresses that are valid for computing the hash value. Finally, the hash is calculated on all the remaining bytes. A detailed description of the algorithm is given in the following section.

7. Checksum Generator Algorithm

We calculate the cryptographic hash of a running process, as there can be changes in the assembly code of the PE file that is running as compared to a static exe. We consider the .text section of the process, as it contains the programmable code. Each process running in memory has a Process ID, which is the unique ID of each and every process. Once the tool is given an input of the PID it performs calculations on the code section, and the end product is a cryptographic hash of that process. Since the process is running on a system that has ASLR present in it, every time the process is run and the exe is loaded into a different memory address there is a difference in the cryptographic hash value.

The algorithm starts by asking the user about the PID of the process for which the user would like to calculate the hash. When the user provides the PID as input, the tool gets

the snapshot of the process with the Windows functions "`CreateToolhelp32Snapshot`"; it also gets the handle of all modules related to that process. The handle of the module is populated with the entry module from the "`MODULEENTRY32`" structure. This helps in getting the various attributes like the Name, Path, and Image Base of the process.

Now we get the handle of the process for further computation. Here we use another Windows function "`OpenProcess`", which gives the handle of the local running process. After this we read the bytes of the process we are interested in. We get all the bytes of the process and save this information in a buffer using the structures IMAGE_DOS_HEADER and IMAGE_NT_HEADERS with the function "`ReadProcessMemory`". The PE file's first byte begins with the DOS headers, and the PE header file offset is stored in this. The IMAGE_NT_HEADERS structure stores information about the PE file – basically, the IMAGE_NT_HEADERS structure has two more fields, IMAGE_FILE_HEADER and IMAGE_OPTIONAL_HEADER, and these two structures contain more information about the PE file, such as,

1. Number of Sections
2. Time Date Stamp
3. Pointer to Symbol Table
4. Address of Entry Point
5. Base of Code
6. Size of Image [13].

After getting all the information, the algorithm now loops through the section headers to find the section that contains the programmable code, and gets it into a buffer.

An assembly instruction can be any size between one and fourteen bytes, but always has the same basic six-part structure:

1. Prefixes (0 to 4 bytes) – optional
2. Opcode (1 to 2 bytes)
3. ModR/M (1 byte)
4. SIB (1 byte)
5. Displacement (1 byte)
6. Immediate (1 byte) [14].

To compute the hash, the algorithm now loops through the bytes saved in the buffer – i.e., the bytes from the code section – one instruction at a time. We start with the first four bytes; if it has relocation we skip these bytes, if not we put these bytes into a buffer, which we can call hashBuffer, for now. After the first four bytes have been put into the hashBuffer, we move the pointer to the next four bytes and then again check for four bytes; if these bytes have relocation, we skip these bytes. Otherwise, we put these next bytes in the hashBuffer. As such, we keep increasing the pointer every time, comparing the bytes of relocation. This way, by the end of the code section we are left with only those bytes that are safe for computing the hash.

We check the relocation by looking at the address – if the address is in the range of (image_base) and (image_base + size_of_section), then it has been relocated to a different address, and we do not consider this part of the instruction.

For example, if there is an instruction

mov eax, [address]

then since this is loaded into memory in the presence of ASLR, this instruction when loaded into memory becomes

mov eax, [address + offset],

or, we can call it

mov eax, [new address].

This means "[new address]" will be a relocated address. Next, we check if this new address lies in the range of (image_base) and (image_base + size) – if it does not exist in this range we skip this "new address" and put the rest of the bytes in the buffer the checksum of which is to be calculated.

8. Testing and Results

After creating our checksum generator, the following tests were performed to check the integrity of the generator. We tested it with various exe's, not only against non-system exe's, but also against system exe's like notepad.exe, calc.exe, paint.exe, and so forth.

Below is a detailed description with results of the testing of the checksum generator with non-system and system exe's.

8.1. Test on regular PE files

In this section, we check our algorithm with non-system exe's that we created, using the option in Visual Studio 2008 to create an ASLR-compatible exe by enabling the option /dynamicbase. We created a sample "Hello World" program – this program simply prints "Hello World!" and shows as a running process in the task manager.

Test Case 1 – helloworld.exe

First Run

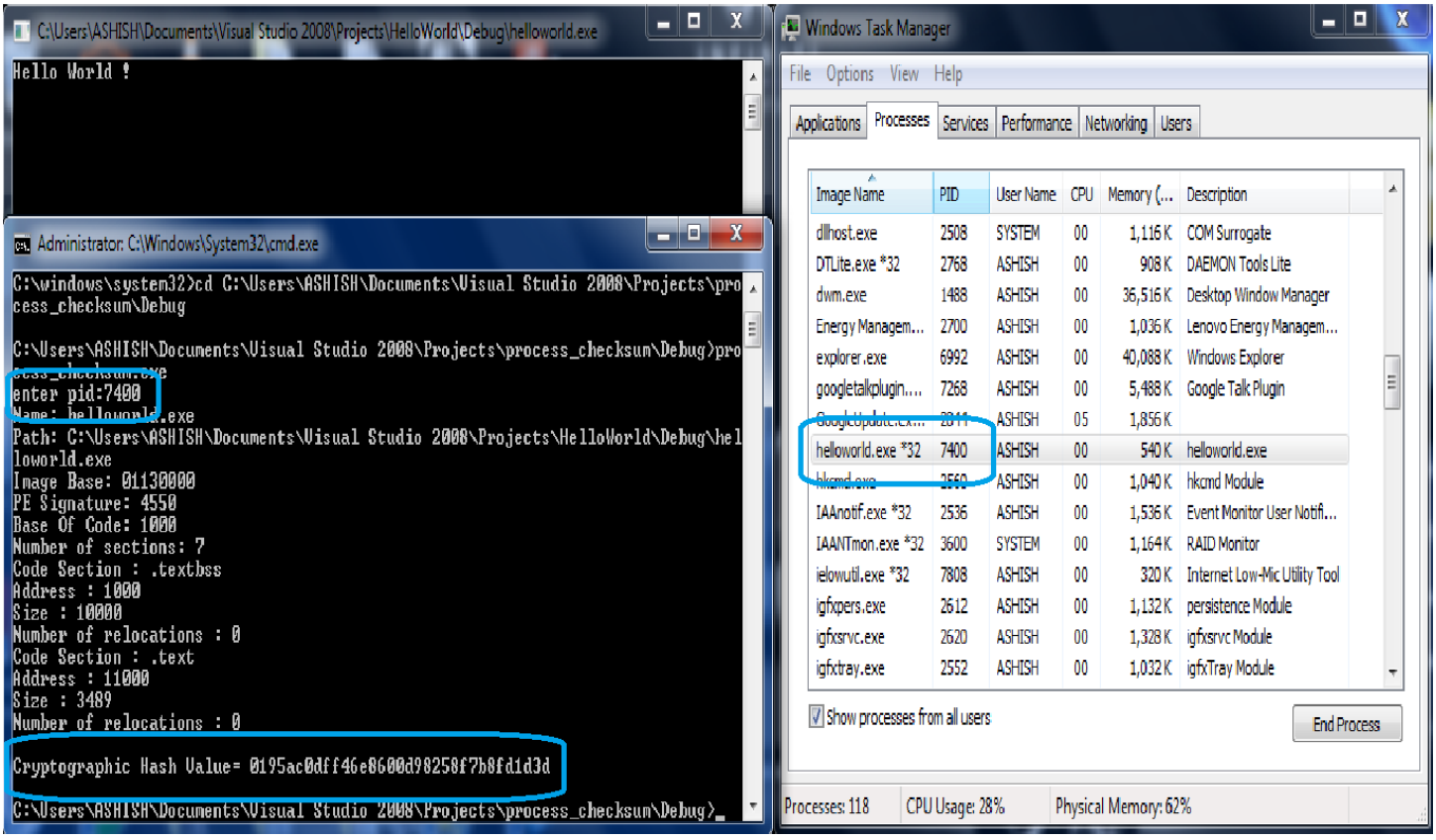

Figure 5: First run for Hello World application.

As we can see, our checksum generator calculates the hash value for the Hello World application as "**0195ac0dff46e8600d98258f7b8fd1d3d**" with PID as "**7400**."

This was the first run of the application. Next, we close the application and restart it so that it has a different PID, and our tool calculates the hash value again. The hash value in the second run should be the same as the hash value in the first run. Also, the PID of the process for the second run should be different from the PID of the process for the first run.

Second Run

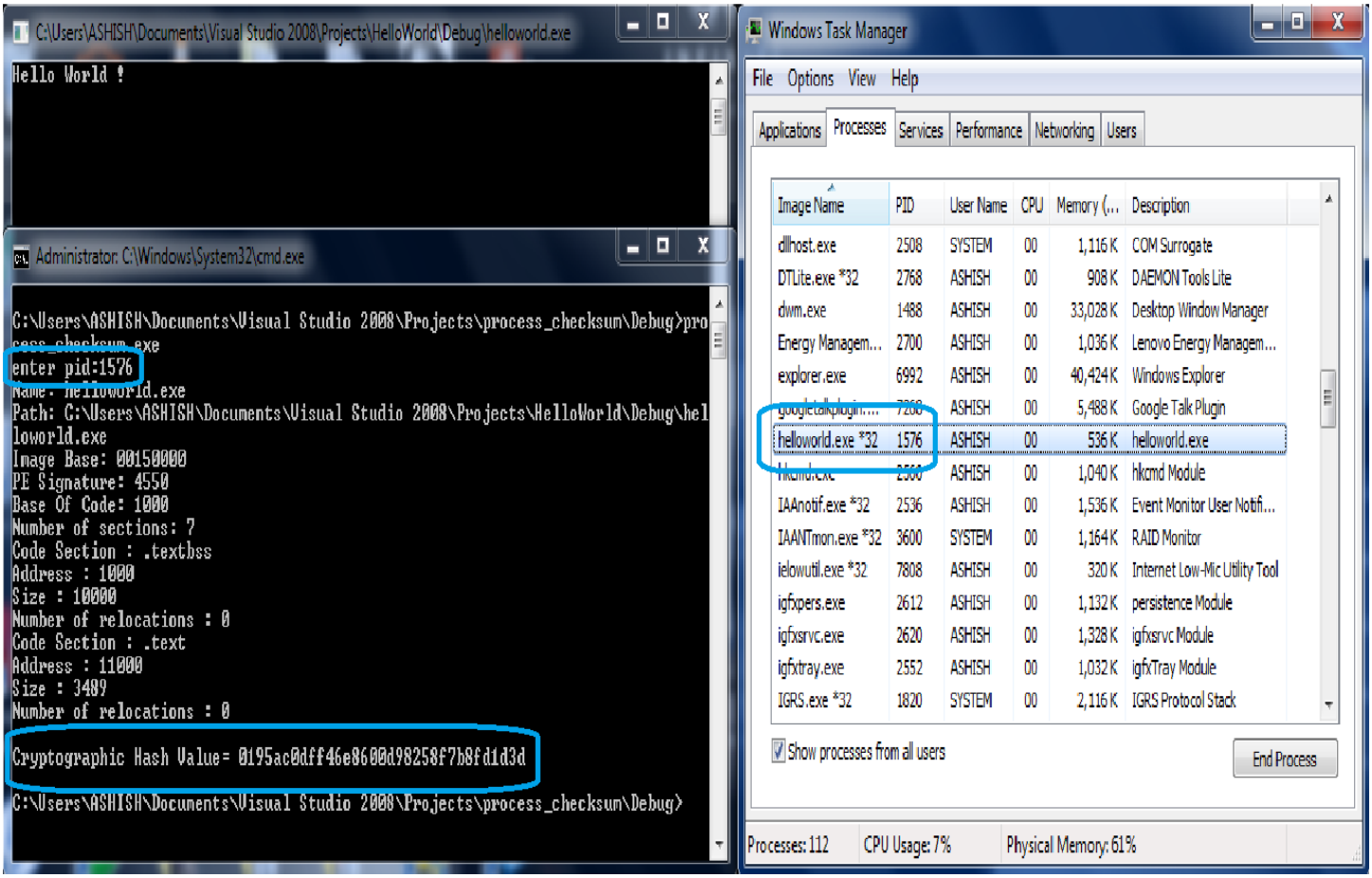

Figure 6: Second run for Hello World application

For the second run, the cryptographic hash value of the running application helloworld.exe is "**0195ac0dff46e8600d98258f7b8fd1d3d**" and the PID is "**1576**."

When we compare the hash values of the same application but for two different iterations,

Cryptographic Hash value (first run): **0195ac0dff46e8600d98258f7b8fd1d3d**

PID (first run): **7400**

Cryptographic Hash value (second run): **0195ac0dff46e8600d98258f7b8fd1d3d**

PID (second run): **1576**

we can see that both hash values are the same, but the process ID is different for each case.

Similarly, the checksum generator algorithm was tested on various dummy applications, and the result was the same: an identical hash value, but a different process ID number.

8.2. Test on System Exe's

Test Case 2 – notepad.exe

Another scenario is to test the tool on system exe's, like notepad.exe, paint.exe, or calc.exe. For this test case, we have selected notepad.exe.

First Run

First we opened an empty Notepad file, and then we gave our algorithm an input of the PID of Notepad.

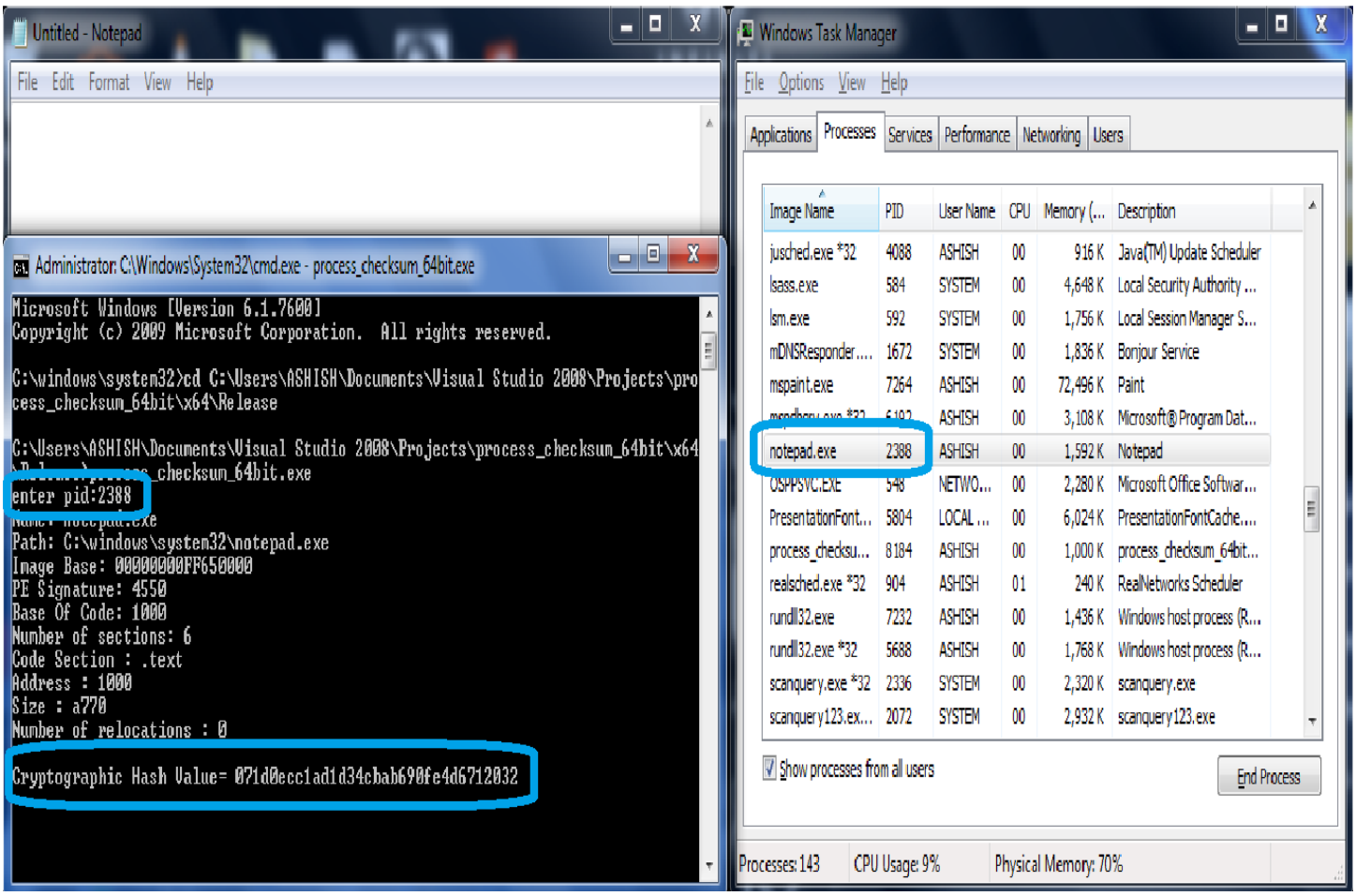

Figure 7: First run for Notepad application

As we can see, the tool calculates the hash value for the Notepad application as "**071d0ecc1ad1d34cbab690fe4d6712032**" with a PID of "**2388**."

Now, as we did earlier, we close the exe and then open it again to get a different process ID.

Second Run

We have the hash values for the first run. Now we can compare them to the values for the second run.

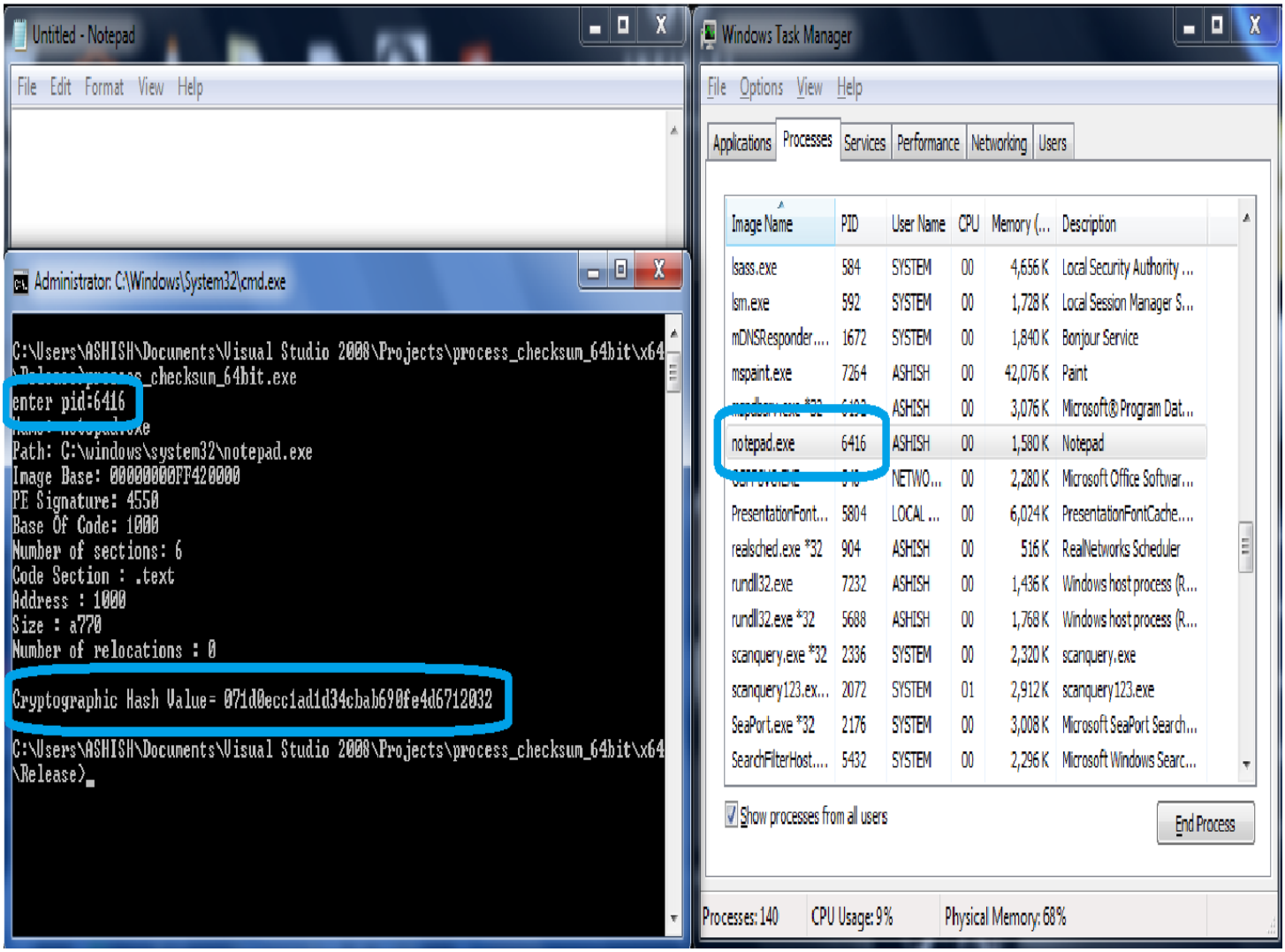

Figure 8: Second run for Notepad application

The result of this run is as follows:

Cryptographic Hash value = **071d0ecc1ad1d34cbab690fe4d6712032**

PID = **6416**

Third Run (on a different machine)

Just to be certain that our tool is robust enough and can produce the same result on different platforms, we ran our tool on another system. Below is the screenshot of the result:

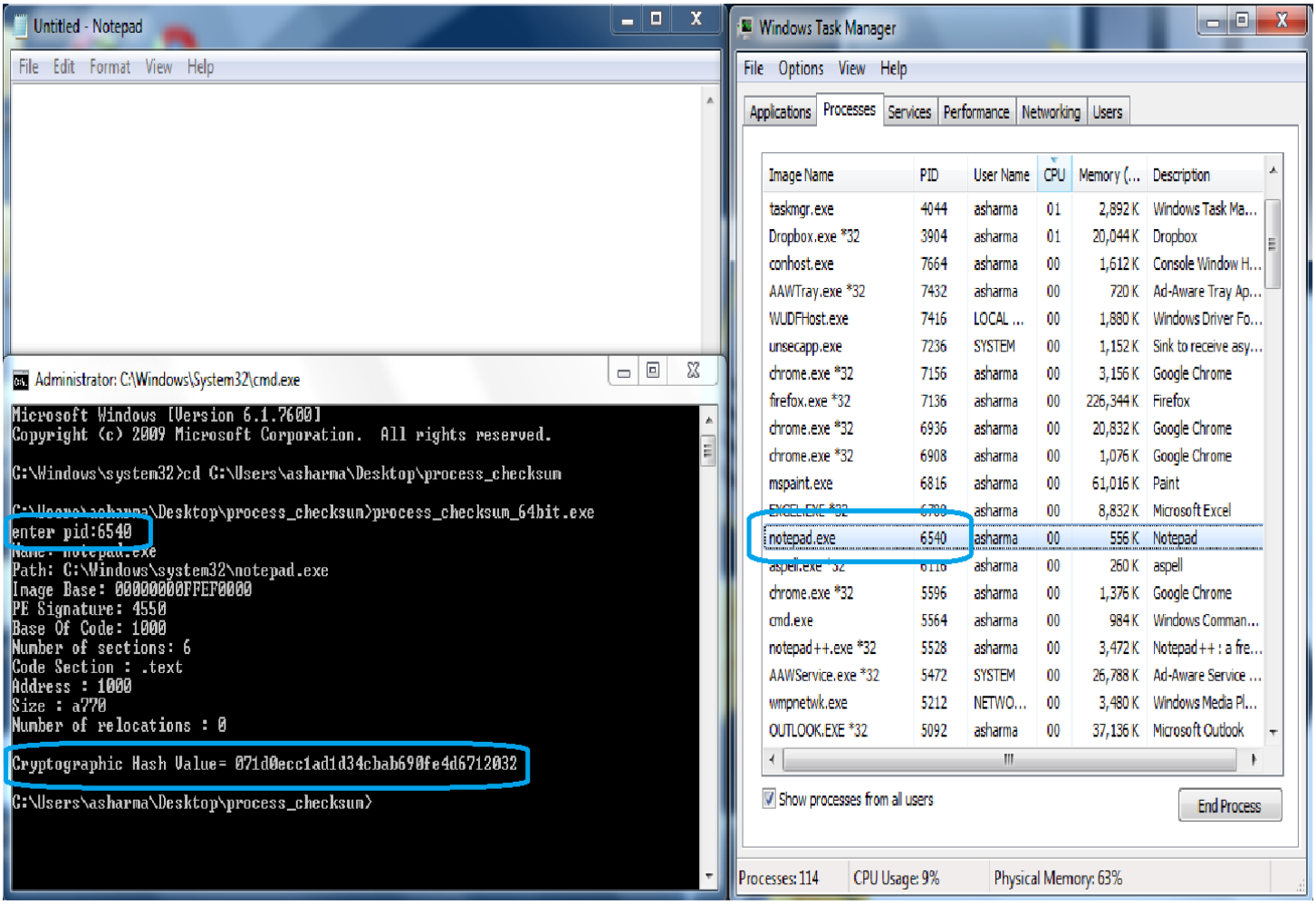

Figure 9: Third run for Notepad application

The result from this run is as follows:

Cryptographic Hash value = **071d0ecc1ad1d34cbab690fe4d6712032**

PID = **6540**

Now we can compare the results from each run.

1. Cryptographic Hash value (first run): **071d0ecc1ad1d34cbab690fe4d6712032**

   PID (first run): **2388**

2. Cryptographic Hash value (second run): **071d0ecc1ad1d34cbab690fe4d6712032**

   PID (second run): **6416**

3. Cryptographic Hash value (third run): **071d0ecc1ad1d34cbab690fe4d6712032**

   PID (third run): **6540**

As we can see, the hash values of Notepad for both runs on the same machine are identical. The value is also the same for the third run, which was done on a totally different machine but produced the same result. However, the process ID is different in all cases.

Test Case 3 – Calc.exe

We also ran another test on a system exe file, this time calc.exe.

First Run

Below is the snapshot from the first run:

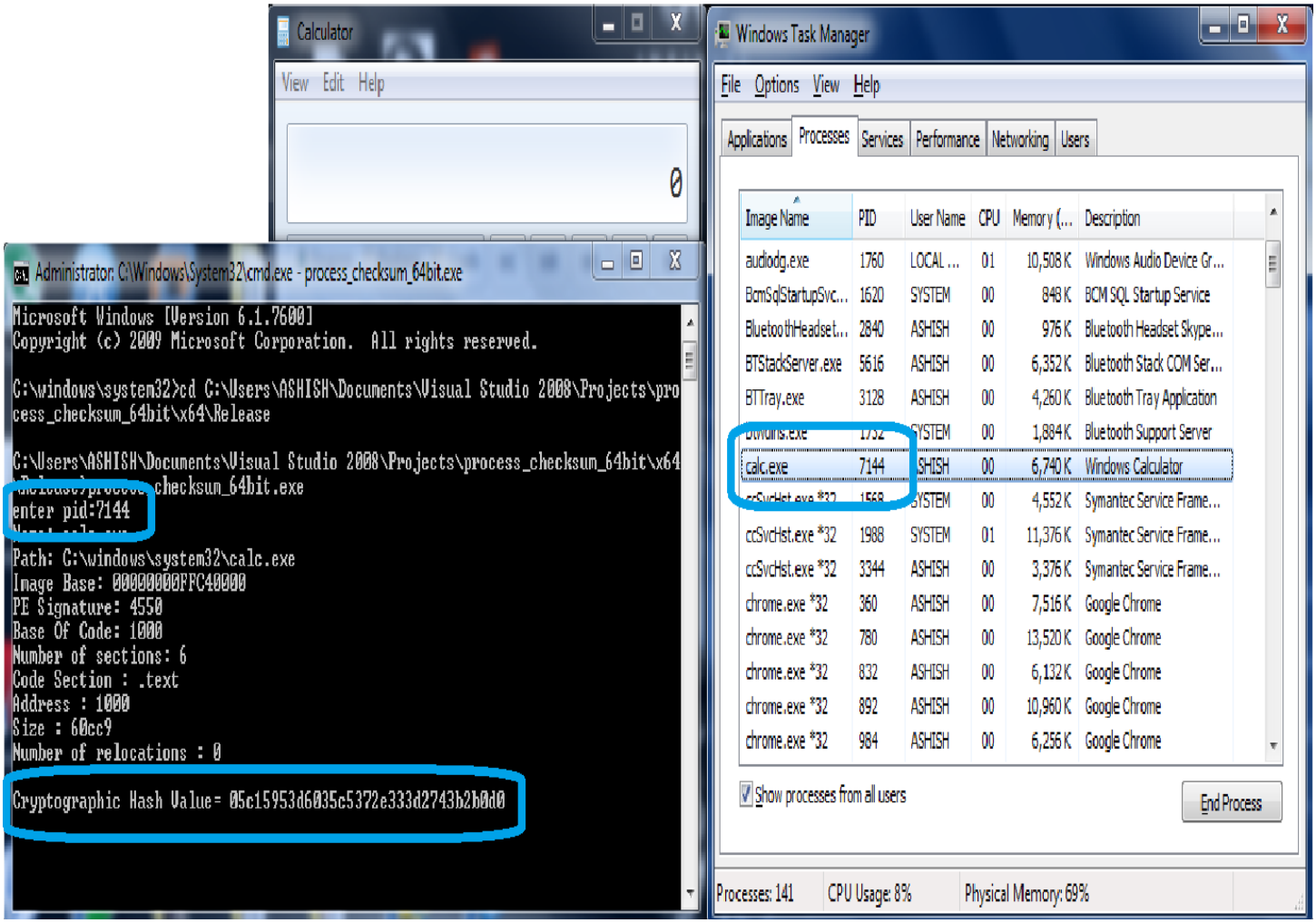

Figure 10: First run for calc application

Below are the results of the first run:

Cryptographic Hash value = **05c15953d6035c5372e333d2743b2b0d0**

Process ID = **7144**

Second Run

We ran calc.exe for a second time, and again collected the results.

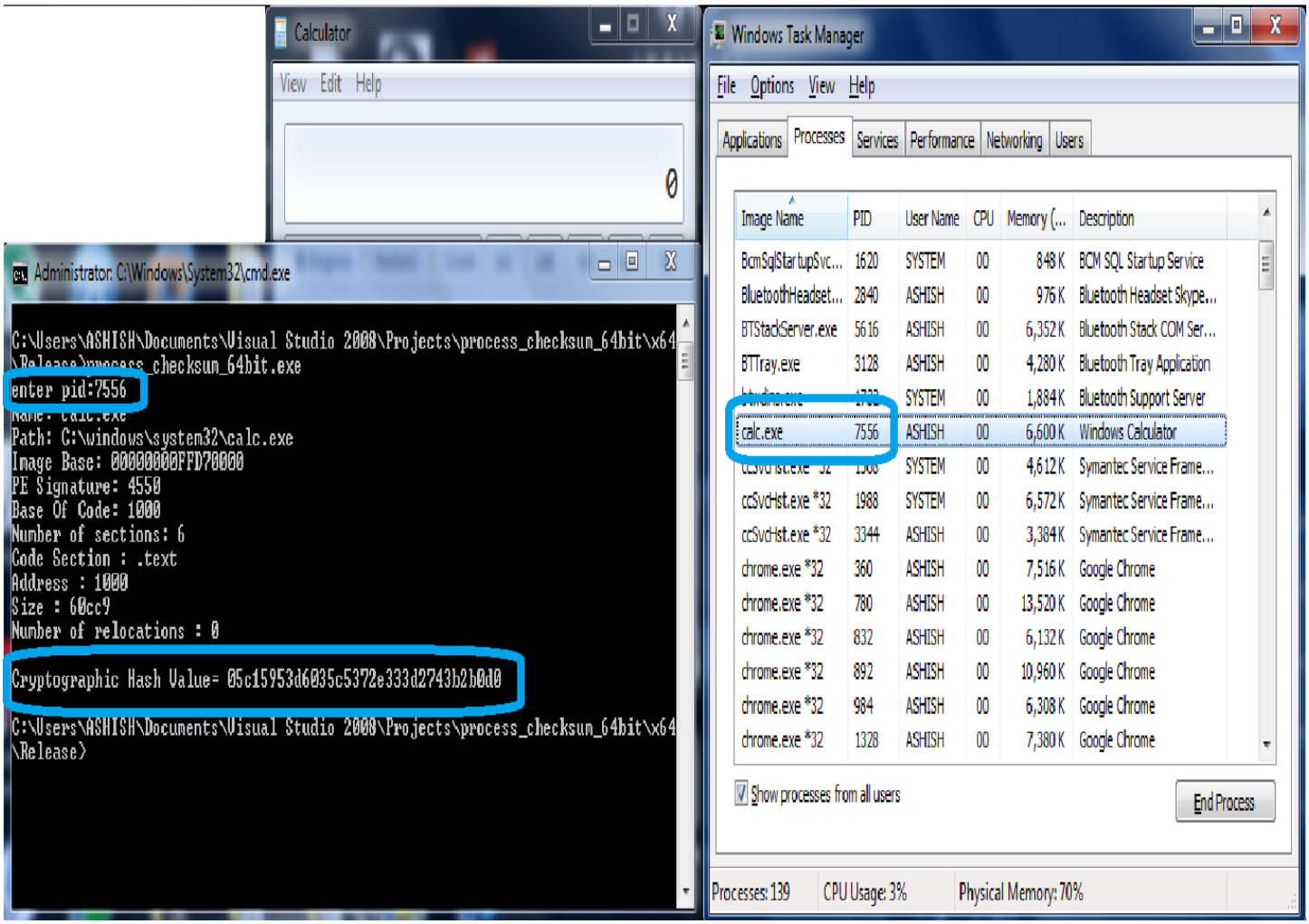

Figure 11: Second run for calc application

The results after the second run are,

Cryptographic Hash value = **05c15953d6035c5372e333d2743b2b0d0**

Process ID = **7556**

When we compare the two results, we see the following:

Cryptographic Hash value (first run): **05c15953d6035c5372e333d2743b2b0d0**

PID (first run): **7144**

Cryptographic Hash value (second run): **05c15953d6035c5372e333d2743b2b0d0**

PID (second run): **7556**

8.3. Performance Analysis Testing

To check the performance efficiency of the checksum generator, we calculated the checksum of an application both with and without relocation. In the "with relocation" scenario, the checksum generator skipped the address relocations; in the "without relocation" scenario it did not skip any relocations, which means it calculated the checksum of all the bytes in the .text section. This test was run 10 times on a 32-bit operating system. The figure below shows the time difference between the two algorithms: The x-axis represents the number of times the application was run, and the y-axis represents the execution time taken by the application in milliseconds. As we can see in the graph, the difference between the two algorithms is not enormous – the average time difference between the two algorithms is **0.0101 milliseconds**, which is not a very substantial value. So, we can conclude that the checksum generator does not make a meaningful impact on the efficiency of the algorithm.

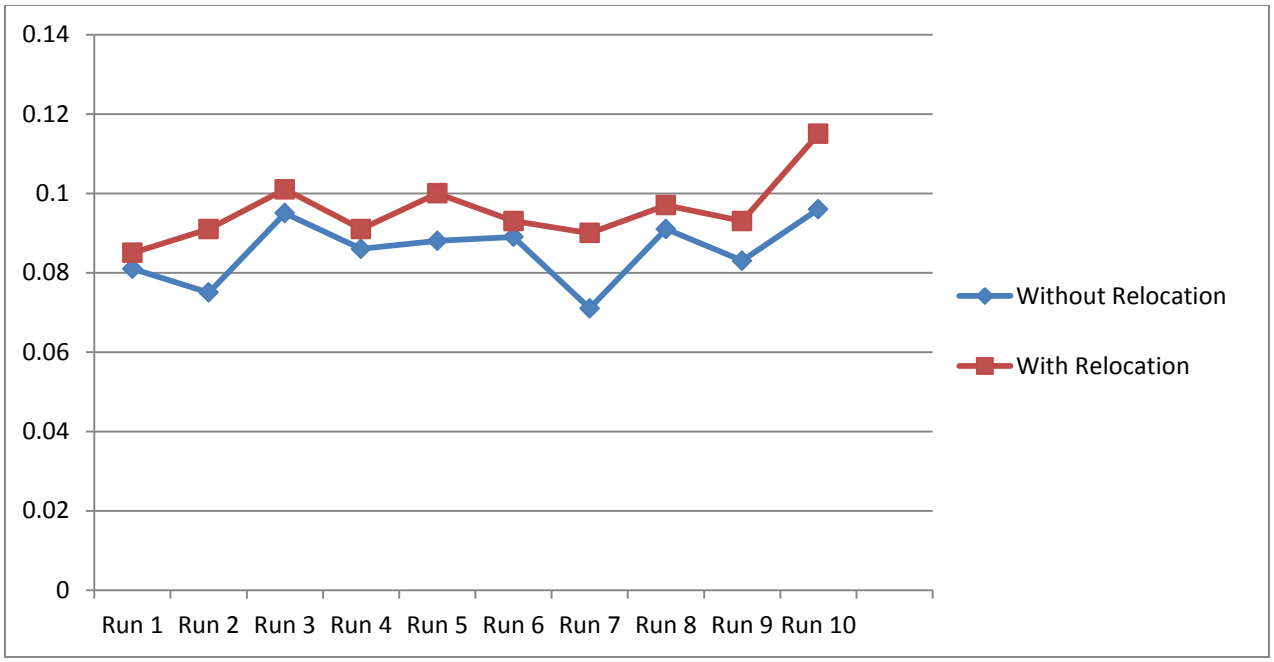

Figure 12: Efficiency of the algorithm

To check the percentage of addresses relocated, we iterated through all of the addresses in the text section of helloworld.exe:

Total number of addresses parsed = 19441

Total number of address relocations = 229

Hence, the percentage of addresses that were relocated for the Hello World application came out to be **1.18%**.

We performed the same test on another application, aslr_test.exe; the purpose of this application was simply to print numbers in an increasing order.

Total number of addresses parsed = 21134

Total number of address relocations = 280

Percentage of relocation addresses = **1.32%**

For checking the robustness of the application we also performed another test, running the anti-debugging of the Hello World application using IDA Pro [24]. During anti-debugging we applied various breakpoints in the .text section, and in one of the instructions we changed the name of a register from ESI to EAX. This change was done when the application was running as a process in the memory and was assigned a process ID.

Checksum for normal Hello World = **0195ac0dff46e8600d98258f7b8fd1d3d**

Checksum for changed Hello World **= 03cb9d9d6401c907a900cf4d306acbbd0**

So, as we can see, if there is even a small change in the application, the checksum will change. This is a good thing, as it allows comparison of the checksum at regular intervals – even if an attacker tries to make changes during anti-debugging, we can seize the application.

8.4. Testing on different applications

After running three different exe's, we find that the algorithm works fine, and as expected gives the same cryptographic hash values for different process ID's.

To further demonstrate the ability of the algorithm, we ran the checksum generator on additional randomly-selected executables, including both system and non-system files. The results of the checksum generation are shown below:

Application Name	First Run PID	First Run Checksum	Second Run PID	Second Run Checksum
MS Paint	4636	0c331146f61d2433a2313d9857397cffa	3848	0c331146f61d2433a2313d9857397cffa
Sticky Note	9596	0028ceeae82c45c9cf9890d60bce74f7f	3020	0028ceeae82c45c9cf9890d60bce74f7f
Edit Plus 3	9932	00eb71b558f92e75948192734db27f450	6096	00eb71b558f92e75948192734db27f450
Hearts game	6004	063218e8e059a761e43a5287433982e90	3448	063218e8e059a761e43a5287433982e90
Google Talk	5080	0d41d8cd98f00b204e9800998ecf8427e	2028	0d41d8cd98f00b204e9800998ecf8427e
Real Player	8608	0e50a9345de926db85e00c09ebf7b7236	9152	0e50a9345de926db85e00c09ebf7b7236
Yahoo Messenger	7124	04f23e1102306c5c9dba9dbd070e2ea02	8896	04f23e1102306c5c9dba9dbd070e2ea02
Word Web Pro	2236	077a53e8d5ad0406e6270e69274df56ba	8412	077a53e8d5ad0406e6270e69274df56ba
VLC media player	4976	082c2dcf2bf8544c00cae85c069961101	6328	082c2dcf2bf8544c00cae85c069961101
Digsby	2216	0043244d26b465fb1ae13a71d6de96af0	7248	0043244d26b465fb1ae13a71d6de96af0
WinRAR	7684	0cb5cb5b5d9f005ca2ed313ae7b546f23	8360	0cb5cb5b5d9f005ca2ed313ae7b546f23
QuickTime Player	9204	0b3496d65059485ed6a9e176fd6ceccf8	412	0b3496d65059485ed6a9e176fd6ceccf8
OllyDbg	6784	01a91c57b07e35f6bb9885531db21464c	7616	01a91c57b07e35f6bb9885531db21464c
Graph	7136	0655a583033ec3210179977ff32a5d5e8	6188	0655a583033ec3210179977ff32a5d5e8
LordPE	7464	0b06b1316f3d50f623670fbba2960df29	8640	0b06b1316f3d50f623670fbba2960df29
Home Game Hero v 1.1	9044	08870387966c19f0ba5261c309cb8282a	8916	08870387966c19f0ba5261c309cb8282a
Viz Calculator	4184	0ad5935682e1388f4ed20c271df3677a1	7596	0ad5935682e1388f4ed20c271df3677a1
Icon Changer	7112	0517fee48e413d8dff3dfff84be531c6d	10088	0517fee48e413d8dff3dfff84be531c6d
Keno	244	0530c1b84f34666b54b636219de0e35a6	7296	0530c1b84f34666b54b636219de0e35a6
Blackjack game	4880	022d66afebb54d2892786c90d54b281f0	9752	022d66afebb54d2892786c90d54b281f0
Free youtube to mp3 converter	7908	0cfb33c16f77702d8c46819f0a0a994cd	4852	0cfb33c16f77702d8c46819f0a0a994cd
IDA Pro	8616	08fe229fa38e717c1037e0556c8188a1f	8252	08fe229fa38e717c1037e0556c8188a1f

Picasa 3	9296	0a2a408c6127f0ef324cb8cc5eea992d3	7404	0a2a408c6127f0ef324cb8cc5eea992d3
Wordpad	2168	0b9642445b1513a1fae564d0ca09df62f	8256	0b9642445b1513a1fae564d0ca09df62f
Lenovo Easycapture	7076	0d9b44cebdafebf71e61f05e8dc3600c5	6780	0d9b44cebdafebf71e61f05e8dc3600c5

Table 1: Results of Checksum Generator

9. Conclusion

Our results demonstrate that, irrespective of the randomness introduced by ASLR, it is possible to calculate the checksum of a process that is loaded into memory. This checksum will always remain constant regardless of whether the system is rebooted, or even if the checksum generator is run on a different machine. The most important use of this checksum would be to safeguard the integrity of an application. Calculating the checksum while the process is running ensures that there are no changes in the executable code, and it is therefore safe to run the application.

There has always been a war between the so-called “good guys” and “bad guys” of the computing world. The good guys develop something, and the bad guys try to break into it. No matter what the engineering world develops, there will always be a crack or a patch available to counter it. This leads to a loss in labor and money for the good guys.

This tool described in this report is made only to increase the security of software. This is not a foolproof security measure, but it makes the work of an attacker a bit more complicated, and perhaps adds to his frustration. In the end it is just that one bit, which if found and changed can breach security.

10. Future Work

Future work for this project would lead towards an efficient way to include this checksum generator in an actual application, and then validate the calculated hash value when the application is run. However, this integration of the checksum generator with an application would be of no use if an attacker could make changes in the executable code of the checksum generator itself. As such, future research could be to obfuscate the coding of this generator, so that it is difficult for an attacker to understand the logic behind the checksum generation.

11. References

1. Dr. Mark Stamp, Chapter 11 Software Flaws and Malware, Chapter 12, Insecurity in Software, Information Security, Principles and Practice (2006).

2. Microsoft Portable Executable and Common Object File Format Specification, 2008.

3. An Analysis of Address Space Layout Randomization on Windows Vista™. Symantec Corporation, Ollie Whitehouse, 2007.

4. On the effectiveness of Address Space Randomization, Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, Dan Boneh. http://benpfaff.org/papers/asrandom.pdf

5. PSAPI Functions, Msdn, http://msdn.microsoft.com/en-us/library/ms684894(VS.85).aspx

6. Portable Executable, Wikipedia, http://en.wikipedia.org/wiki/Portable_Executable

7. Attacks on Win 32, Part II, Peter Szor. http://www.symantec.com/avcenter/reference/attack.on.win32.pdf

8. CIH Computer Virus, Wikipedia, http://en.wikipedia.org/wiki/CIH_virus

9. F-Secure Virus Descriptions 5lo Virus, http://www.f-secure.com/v-descs/5lo.shtml

10. Image Section Header Structure, Msdn, http://msdn.microsoft.com/en-us/library/ms680341(v=vs.85).aspx

11. Cryptographic Hash Function, Wikipedia, http://en.wikipedia.org/wiki/Cryptographic_hash_function

12. Digital Signature, Wikipedia, http://en.wikipedia.org/wiki/Digital_signature

13. In depth look into Win 32 PE file format, Matt Pietrek, 2002, http://msdn.microsoft.com/en-us/magazine/cc301805.aspx

14. Assembly Instructions Format, William Swanson, 2003, http://www.swansontec.com/sintel.html

15. Virus Encyclopedia, Win32.IKX virus, http://www.virus-encyclopedia.com/virus/Classic-Viruses/File-and-Boot/virus456.html

16. Virus Database, Win 95.Invir.7051 virus, http://www.virus-database.com/description/5023-win95.invir.7051.html

17. Integrity Checking, VX heavens, http://vx.netlux.org/lib/static/vdat/glossary.htm

18. Address Space Randomization, for Windows system, Lixin Li & James E. Just, http://seclab.cs.sunysb.edu/seclab/pubs/acsac06.pdf

19. Total Electronic Migration System, newsletter for Information Assurance Technology Professionals, 2005, http://iac.dtic.mil/iatac/download/Vol7_No4.pdf

20. Linux ASLR Vulnerabilities, Jake Edge, 2009, http://lwn.net/Articles/330866/

21. Security Research and Defense, On the effectiveness of DEP & ASLR, 2010, http://blogs.technet.com/b/srd/archive/2010/12/08/on-the-effectiveness-of-dep-and-aslr.aspx

22. Evolution of 32 bit Windows Viruses, Peter Szor & Eugene Kaspersky, 2000, http://vx.netlux.org/lib/static/vdat/epevol32.htm

23. OllyDbg disassembler and debugger, http://www.ollydbg.de/

24. IDA Pro Disassembler and Debugger, http://www.hex-rays.com/idapro/

25. Introduction into Windows Anti debugging, Josh Jackson, 2008, http://www.codeproject.com/KB/security/Intro_To_Win_Anti_Debug.aspx?display=Print

26. Static Analysis of Binary Executables, Steve Hanov, http://stevehanov.ca/cs842_project.pdf