DIME: Distributed Independent Malware Execution

A THESIS

submitted by

JITHIN PAVITHRAN

in partial fulfilment of the requirements

for the award of the degree of

BACHELOR OF TECHNOLOGY

and

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

MAY 2019

THESIS CERTIFICATE

This is to certify that the thesis titled **DIME: Distributed Independent Malware Execution**, submitted by **JITHIN PAVITHRAN**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor & Master in Technology (Dual Degree)**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Chester Rebeiro
Research Guide
Assistant Professor
Dept. of Computer Science
IIT Madras, 600 036

Prof. Nitin Chandrachoodan
Research Co-Guide
Associate Professor
Dept. of Electrical Engineering
IIT Madras, 600 036

Place: Chennai

Date: 5 May 2019

ACKNOWLEDGEMENTS

I express my sincere gratitude to my project guide Prof. Chester Rebeiro for giving me an opportunity to work in the field of cyber-security. His advice, encouragement and co-operation throughout the course of my work hold a significant role in the completion of this work. I would like to express my gratitude to Col. Milan Patnaik for introducing me to this wonderful field of malware studies. Interactions with him have been invaluable in developing my interest in this field of research. I also like to thank Prof. Nitin Chandrachoodan for his supervision as co-guide for my project.

I like to thank Signal Intelligence, Govt. of India for their valuable support in my research. I also thank RISE Lab for providing a wonderful, vibrant work environment. I also like to give credit to my beloved institute, Indian Institute of Technology, Madras for providing me with every resource required for the completion of this project.

ABSTRACT

KEYWORDS: Malware; Obfuscation Techniques; Behavioural Detection; Heuristic Detection; Distributed Execution

Fighting malware is crucial to secure computer systems. Malwares are designed to be stealthy on their target to remain undetected. The cat and mouse game of obfuscating and detecting the malware has been played for a long time. After the invention of behavioural and heuristic detection techniques, not many successful obfuscations techniques were developed.

In this thesis, we introduce a Distributed Independent Malware Execution framework (DIME) to evade behavioural and heuristic malware detection techniques. DIME distributes the malware execution across benign threads in the system. The framework run without creating any new thread or process. The distributed decentralised execution of the malware successfully evades behavioural as well as heuristic based malware detection.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

DIME	Distributed Independent Malware Execution
SCBC	Semaphore based Covert Broadcasting Channel
APC	Asynchronous Procedural Call
API	Application Program Interface
OS	Operating System
IO	Input Output
FIFO	First In First Out
ROP	Return Oriented Programming
BBS	Basic Block Splitting
BAST	Below AV Signature Threshold
PoC	Proof of Concept
CFG	Control Flow Graph

CHAPTER 1

INTRODUCTION

1.1 Malware

Malware attacks are a serious threat to the security of computer systems. Malware is an umbrella term used to describes a piece of software that is harmful to systems. The harm could be caused to the software, data, user, network or even the infrastructure itself.

Malwares are commonly categorized based on their propagation method and functionality [9].

- *Viruses:* These are malwares that replicates themselves. These programs replicate within a host by attaching themselves to programs and/or documents. The program/document on which the virus replicates is called the carrier. A transit of the carrier documents causes the virus to spread to other computers as well.
- *Worms:* Worms are malwares that spread over computer networks. They replicate themselves and infects computers over networks.
- *Trojan Horses:* A Trojan horse or *Trojan* is malicious software that disguises itself as legitimate. In most cases, they provide useful functionality to the user while carrying out malicious intent in the background.
- *Backdoors:* This type of malware open their victim to external entities. Backdoors help the external entity (the hacker) to remotely control the system.
- *Spyware:* A spyware transmits or leaks the user data to an external entity. Unlike the Trojan, spyware runs without the knowledge of the user.

A Malware may not necessarily exploit security vulnerabilities of a system. Rather they are the payloads executed on the victim post-exploitation. Malwares are widely used by hackers to maintain access to compromised systems. Naturally, evading detection is of utmost importance. In most cases, early detection of malware would result in

mission failure. It will also waste the attacker's efforts and resources to compromise the system. Thus malwares are designed to be stealthy on their targets. While hiding themselves is important for malware, detecting the malware becomes equally important for the defenders. The battle of hiding and exposing malware has been fought right from the invention of computer viruses. As a general concept in cybersecurity, the battle can only be won if one starts to think from the other side.

A variety of techniques has been deployed by malware authors to hide their software from anti-malware scanners. Primitive malware detection was done by static scanning of executable. They looked for patterns that could be recognized as malicious. Attackers quickly devoted themselves in static obfuscation techniques. They encrypted the malware and added a decryptor in the executable. The malign code would be decrypted at run-time for execution. Naturally, defenders got suspicious of executables with encrypted content. They looked for encrypted sections in the binary and detected the decryptor. Techniques like *Movfuscator* and *Virtualized malware execution* evolved to make detection harder. Movfuscator compiles a program to contain only the `mov` instructions. In virtualization based techniques, attackers compiled the malware for their custom architecture. Emulators were used for virtualizing their architecture to enable malware execution. Analyzing an executable obfuscated with these techniques was difficult. But detecting the use of such techniques was trivial. Thus new categories of malware such as polymorphic and metamorphic malware were invented. These malware programs were able to change their own structure without affecting their functionality. Static detection was found to be ineffective at these obfuscation techniques. Behavioral and Heuristic analysis techniques were invented.

Behavioral and heuristic detection techniques analyze a program at run-time to guess if it is malign or not. They look for Behavioral signatures such as patterns in system calls made. Software are often executed in *test-environments* to study their run-time behavior. These studies help to create Behavioral signatures. To counter Behavioral and heuristic detection techniques, attackers came up with a variety of approaches. Platform aware malwares were built. These programs would detect test environments and behave benignly in them. This fools the analyzer or complicates the analysis. Attackers use process injection techniques to execute a malign thread under a legitimate

process. This camouflaging technique hides the malign behavior under victim process execution. It also helps overcome application based permission models introduced by OSs like Windows, Android and Mac OS X 10.9. Inserting unused system calls and sleep statements are other simple techniques used to evade behavioral detection. Anti-virus programs defend these techniques by isolating the program execution and filtering out the fictitious statements.

1.2 Introduction to DIME

In this thesis, we introduce DIME, a Distributed Independent Malware Execution framework designed to evade behavioral and heuristic detection. We distribute the malware execution across benign threads in the system. Behavioral and heuristic analysis fail at our distributed malware execution. We also introduce SCBC, a new covert channel based on Semaphores. Though the covert channel was originally invented to support requirements of DIME, it has a much broader scope.

An earlier attempt to distribute malware execution is found in MalWASH [16]. MalWASH created multiple threads in benign processes and distributed the malware across these threads. They required administrative privileges for execution. However, DIME does not create any new threads in the system and execute with the normal user privileges. It escalates the detection difficulty and urges the invention of new classes of detection methods. It leaves no trackable behavioral signature in the system. DIME also offers high resilience, making the removal of the malware extremely complicated.

The input to DIME is a malware executable. DIME splits the executable into small instruction sets. These instruction sets are called *chunks*. The splitting activity is done offline. In the target system, we infect and spreads through pre-existing benign threads in the system. We exploit the infected threads to execute one or more of our chunks. Collectively, the infected threads will execute all the chunks. We coordinate these executions in a decentralized fashion. This maintains the essential order and continuity in the execution of chunks. A well coordinated execution of chunks results in the desired malign action. By distributing the chunk-execution we distribute our behavioral signature. The existing malware detection mechanisms fail to detect this distributed

malignity.

The behavioral and heuristic detectors fail DIME since they will not observe any behavioral patterns they are looking for. By distributing the instructions to multiple threads, DIME splits the malware execution patterns and spreads it across space and time. This increases the volume of infection and reduces the malign density of infected space. The chunks are small in size. They get buried in the normal execution of benign threads. To analyze the behavior of DIME, the detector has to first track the chunks back to DIME. The DIME architecture complicates any attempt to do so. DIME is a completely decentralized framework. Its components do not carry any common identifiers. It is capable of infecting almost all processes. In addition, the execution timings of chunks are randomized using intrinsic randomness of the system. This makes any efforts of isolation extremely complicated or impractical.

The DIME architecture is developed based on a Windows feature called APC (Asynchronous Procedural Call). APCs are functions executed in the context of a target thread asynchronously. DIME constructs a completely decentralized execution framework using APCs. The DIME function that gets executed as APC is called *emulator*. Emulators provide a "safe" environment for the chunks to execute. They also coordinate with other emulator executions to maintain the required order of execution of chunks. SCBC, the new covert channel was invented to support this coordination. Emulators are also self-regenerating. i.e. they cause the execution of more emulators in other benign threads. This ensures the continuity of DIME. In addition, this also adds to the resilience of the framework. Having a single emulator in the system is sufficient to bring back the whole DIME.

1.3 Contributions

Following are the major contributions of our work:

1. We propose the first fully distributed, decentralized malware execution framework on a host.
2. We introduce a novel way of exploiting APCs for a thread-less execution of malware.

3. We introduce a new robust covert channel for OSs.

CHAPTER 2

Background and Related Work

2.1 Background

In this chapter, we discuss the essential concepts and technologies required to understand the chapters that follow. We also go over the important detection as well as evasion techniques used in the wild.

2.1.1 Asynchronous Procedural Calls (APC)

An Asynchronous Procedure Call (APC) is a function that executes asynchronously in the context of a particular thread. Every thread in Windows maintains a queue called APC-queue. Any thread in the system can queue a function in the APC-queue of a target thread. The function will be asynchronously executed in the context of the target thread.

APCs can be broadly classified as kernel-mode APCs and user-mode APCs. Kernel-mode APCs are generated by the system and gets executed by means of a software interrupt. Device drivers and the IO Manager use kernel-mode APC to execute callback functions upon completion of the requested transaction. This helps to make these transaction requests non-blocking. User-mode APCs are generated by applications. Unlike Kernel-mode APCs, latter doesn't interrupt the thread. User-mode APCs are queued in the APC-Queue of the target thread. They will be unqueued and executed when the thread goes to *Alertable Wait State*. In other words, user-mode APC is a polite way to interrupt a thread. A thread enters an alertable state when it calls the `SleepEx`, `SignalObjectAndWait`, `MsgWaitForMultipleObjectsEx`, `WaitForMultipleObjectsEx`, or `WaitForSingleObjectEx` function. These functions are commonly called by windows APIs internally. They are also used by threads directly.

Once a thread enters an alertable wait state, it will execute all the queued APCs in FIFO manner. The thread will consume all the queued functions before exiting the wait state.

APC functions should have the following format

```
PAPCFUNC Papcfunc;
void Papcfunc(
  ULONG_PTR Parameter
)
{...}
```

They can be queued using the Windows API

```
DWORD QueueUserAPC(
  PAPCFUNC  pfnAPC,
  HANDLE    hThread,
  ULONG_PTR dwData
);
```

In this thesis, we restrict our discussions to User-mode APCs. The techniques discussed can be directly adopted for Kernel-mode APCs but would need superuser privileges.

2.1.2 Semaphore

A semaphore is a shared counter used to synchronize access to a shared resource. Semaphores maintain a count between zero and a maximum. The count is decremented when a thread starts to use the resource and it is incremented when the thread releases the resource. No more threads can access the resource if the count reaches 0.

Windows provides it's own APIs to create and maintain semaphores. Windows's API `CreateSemaphore` allows applications to create semaphores. Applications should name the semaphore at creation. The semaphore-name is used to uniquely identify the semaphore across the system. `WaitForSingleObject` API can be used to gain access to the resource by decrementing the semaphore counter. `WaitForMultip`

`leObjects` can be used to lock multiple resources. The `ReleaseSemaphore` API is used to release the resource and increment the counter.

2.2 Related Works

Though computer malware has a short history, it has been a very dynamic one. Numerous obfuscation and detection techniques have evolved over time. We shall notice a rapid increase in the complexity of these techniques. Starting with the static detection techniques, the techniques have evolved to highly complicated heuristic detection mechanisms. In this section, we will discuss some of the important obfuscation and detection techniques.

2.2.1 Static Obfuscation Techniques

Primitive Obfuscation Techniques

The earliest attempts of malware detection were static signature based. The concept of the signature itself has evolved a long way. Primitive signatures were sequences of assembly instructions that performed a malicious action. A simple search through the executable was performed to find these signatures. Static detection techniques required the malicious signature to be available in their database for cross-checking. This was a major disadvantage of these detection techniques. They suffered heavily from new malwares. New malware programs had to be detected and analyzed manually to create the signature. There were malware programs which successfully spread for years before getting detected.

To counter the detection based on instruction sequences, attackers developed encrypted malware. Encrypted malware keeps malicious code encrypted. The encryption keys are changed to obtain different versions of the same code. These malwares would carry a small decryptor with them. The decryptor decrypts the encrypted code at runtime for execution. Since the encrypted content look random and every instance uses a different key, no signature could be extracted.

Movfuscator, Virtual instruction set based malware, ROP based malware, and Packers are other examples of static obfuscation techniques. `Mov` instruction is proven to be Turing complete [11]. Thus any program can be written using only the `mov` instructions. Movfuscator compiles the whole program to using only the `mov` instructions. This makes the disassembly and analysis of the code extremely difficult.

As the name suggests, ROP based obfuscation techniques use ROP gadgets to build malware [26]. ROP or *Return Oriented Programming* gadgets are small instruction sequences ending with a `ret` instruction. The malware searches shared libraries to find ROP gadgets. The malign code is then built by stitching these gadgets. Since the execution is performed using gadgets, the actual code will not be available in the executable. This complicates static detection techniques.

In virtual instruction set based malware, the attackers design a custom processor with a set of new opcodes. The malware will be compiled to run on this new processor. These malware programs carry a small emulator with them. The emulator emulates the custom processor to provide a virtual platform for the malware to run. It essentially reads the opcode and translates it into host architecture and executes them. Since the scanner wouldn't know the op-code set of the custom processor, the malware code could not be analyzed.

Usage of Packers is a relatively simpler mechanism. Packers work like self-extracting files [22]. They "pack" the file to obfuscate it. During packing, the file is compressed and the sections are rearranged. The obfuscated file is unpacked at the target system in memory. Thus the malicious file never gets to the hard disc. This avoids the risk of getting caught by disc scanners.

These techniques were effective in complicating the analysis of the code and hiding the malign content. But the use of these techniques was well exposed. Any executable with a decryptor is suspected of being malicious. Similarly, programs with only `mov` instructions are guaranteed to use Movfuscator. Since there are no legitimate reasons to use such obfuscation techniques, programs using these obfuscation techniques can be easily flagged.

Oligomorphic and Polymorphic Malwares

Oligomorphic and Polymorphic malwares were results of attempts to hide the decryptor of encrypted malware. These malwares can mutate their decryptor from one generation to the other. Oligomorphic malwares have multiple decryptor loops. These loops will be spatially distributed over the executable. Different combinations of loops are deployed to change the appearance. The distribution helps to reduce the size of the continuous decryptor code. This reduces the chances of creating a practical signature out of them. Oligomorphic malwares can typically generate hundreds of versions of the decryptor. Polymorphic malwares are more advanced. A well written Polymorphic malware can generate countless number of versions of itself. The decryptors constructed by Polymorphic malwares are different semantic versions of the same underlying code. They use techniques like Dead code Insertion, Register reassignment, Instruction substitution, Instruction permutation, Code transposition, etc... [31]

Metamorphic Malware

While Polymorphic malwares hide their code well at first sight, the underlying code is exposed after decryption. Metamorphic malwares were built to overcome this weakness. Metamorphic malwares do not have encrypted code. Instead, they evolve by creating new semantically equivalent versions of their original malign code. These versions look quite different while serving the same functionality. Metamorphic malwares use similar techniques as used by polymorphic malwares [31]. In addition, they use techniques like Subroutine Reordering as well. They are composed of a Disassembler, Code analyzer, Code transformer, and an Assembler. These modules operate on their own executable file to create new versions of the same. Frankenstein [5] is an obfuscation technique that evolved from metamorphic malwares. They create ROP based malwares. Different combinations of gadgets are used to create new versions of the malware. It is theoretically proven that reliable detection of metamorphic malwares is NP-Complete [28]. Practical attempts of detection were also proven to be very inefficient [7]. This urged the development of a new class of detection techniques. *Behavioral* and *heuristic detection techniques* were developed.

2.2.2 Behavioral and Heuristic detection techniques

Behavior-based based detection techniques look at the run-time behavior of the executable to determine if the binary is malicious or not. Heuristic detection methods take help of data mining and machine learning techniques to support behavioural analysis. Behavioral signatures can be built from different aspects of software execution. Sequence of API calls [3] [30] [18], characteristics of OpCodes [24] [23] [25] and CFG (Control Flow Graph) structure [8] [4] [6] are some of the major aspects examined. [12] and [20] also proposed methods which are hybrid of multiple detection techniques. Binaries are executed in isolated environments to study their behavior. These isolated executions help determine the legitimacy of the executable safely. Behavioral signatures are created for the ones found to be malicious.

Operating systems like Windows, Mac OS X and Android has also come up with application-level defensive techniques to fight malware. They impose several restrictions on resource usage by applications. These restrictions ensure that only the safe and essential applications are allowed to access critical system resources such as the network. For example, a calculator application can be blocked from indexing files in the system.

The classes of Behavioral and heuristic detection techniques are researched extensively. Nevertheless, very few publications have come to counter these detection mechanisms. A simple method to hide malign behavior is to add noise. Unwanted instruction sequences and systems calls are added to the malware. The attempt is to hide the actual malign behavior under a huge volume of random instructions and system calls. Code injection techniques are also used to counter behavioral detection. Malicious code is injected into a victim process's memory. New threads will be created under the target process to run on the injected code. This makes the code as well as the thread to look legitimate under the benign victim process. Modern malwares are equipped with the ability to detect test-environments used by defenders to analyze them [13]. They behave benignly in virtual environments and shows their true colors in the actual host system.

Feature-distributed malware [21] was an attempts to overcome the application based

restrictions and associated detection methods. [21] split the malware to components based on functionality. The components will be then injected to appropriate benign processes. For example, the component that sends the data back to the attacker can be injected to browsers or other processes which use network extensively. The components that index files can be injected to file explorer. The technique was effective in overcoming the process based restrictions.

MalWASH [16] is a recent work to counter behavioral and heuristic detection techniques. MalWASH distributes the malware among multiple processes. It chops the malware executable to pieces and distributes the execution of these pieces in multiple processes. MalWASH assumes that the attacker is able to inject code into multiple processes. It further assumes that the attacker can create threads in multiple processes which run the code they provide. This is a reasonable assumption since obfuscation techniques don't need to worry about exploiting the system vulnerabilities. It should also be noted that a number of process injection techniques are being discovered in Windows. The code that each of the malign thread will execute is called *emulator*. The emulator provides a way to execute the pieces of actual malware executable and coordinate the executions. By distributing the execution among emulator-threads under multiple benign processes, the malignity of behavioral gets distributed. This makes it difficult for the scanners to find any malign behavioral patterns in a process.

Though MalWASH claims to distribute the malware among benign processes, the distribution is not complete. If the process envelope is taken off, one can clearly see the dedicated emulator threads. One of the main flows of MalWASH is the requirement of such dedicated emulator threads. The emulator is essentially a platform for the pieces of code to execute. Though the piece of malware code executed changes from emulator to emulator, the basic functions of emulator remain the same. Thus the emulator-threads have a common signature of its own. Since the emulator code is known, this signature could be generated offline. Thus by detecting emulator-threads MalWASH can be detected easily. As soon as the scanners improve their granularity from process to thread level, i.e, if the scanners look at thread behavior rather than the process behavior, MalWASH is guaranteed to get detected. Another disadvantage of MalWASH is that it required Administrative Privileges. This greatly reduces the scope of practical use of

MalWASH.

CHAPTER 3

DIME

The advanced behavioral and heuristic malware detection techniques focus on the activities of processes. They try to classify each process as malign or benign. The success of these detection techniques depends on how well they isolate the malign content present in the system.

We introduce DIME, a malware execution framework designed to evade behavioral and heuristic detection. DIME is a distributed, decentralized, thread-less execution framework. It distributes the malware across benign threads in the system. Given a malicious executable, DIME splits it into to a number of instruction sets called *chunks*. In the target system, DIME exploits the OS to steal brief execution periods from benign threads in the system. A large number of such execution periods put together forms the execution time of our malware. Chunks are executed in the stolen execution periods, in victim threads. The executions are coordinated to meet the required sequentiality as well as continuity of the code. Required virtualization is also provided for the safe execution of chunks in victim threads' contexts. A well-organized execution of chunks in this manner gets the original malicious action done. The chunk-executions are organized in a decentralized manner. Their continuity is also maintained similarly. A chunk execution causes more of such executions in the future. Similar to most of the biological viruses, they rapidly reproduce themselves and infects every thread in the system.

Detecting DIME as malicious is perplexing due to multiple reasons:

1. There is no single entity who is malicious. It is the collective actions of multiple legitimate threads that cause harm.
2. There is no identifier that can be tracked or isolated.
3. The confinement space for the malware is too bloated with low malign density.
4. There is no central controlling agency. The system is completely decentralized.

Applying behavioral and heuristic detection techniques on DIME is extremely complicated or impractical.

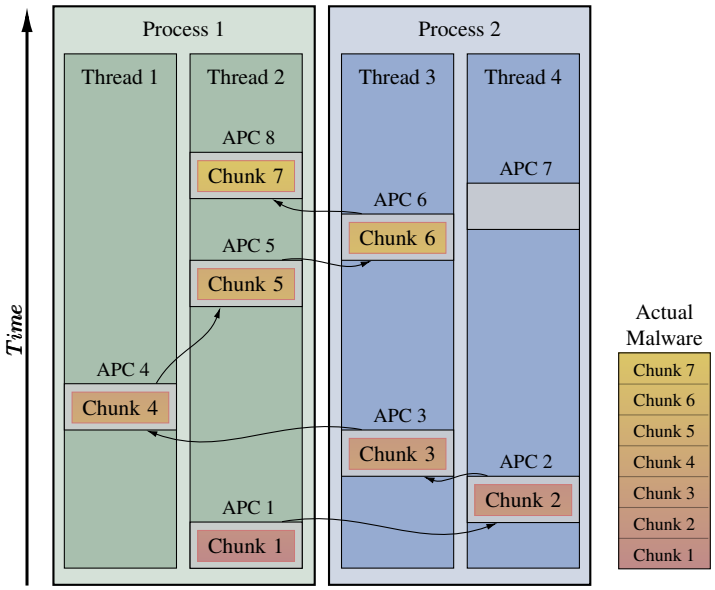


Figure 3.1: DIME: Chunk execution over time.
Chunks getting executed in benign processes as APCs. The time is on vertical axis.

3.0.1 DIME Architecture

DIME architecture is the combination of chunk-executions and their coordination. We use APCs to realize chunk-executions. DIME queues an emulator function as APC in legitimate threads in the system. Emulators execute the required chunk in the thread by coordinating with other emulators. DIME-emulators are regenerating. i.e., they regenerate themselves to get executed in other threads in the future. They look for other threads in the system and queues itself to the APC-Queue of threads. Emulators can Queue itself on threads of the same or a different process. This ensures the continuity of DIME. As long we have at least one emulator in the system, continuity of DIME is guaranteed. This also adds to the resilience. To remove DIME from a system, the defender will have to remove all emulators from every thread simultaneously. Any attempt of sequential removals will fail since existing emulators will queue themselves back.

Figure 3.1 illustrates DIME. The execution begins with APC 1 and continues with the APCs that follow. We cannot exactly predict when a queued APC is going to get executed. They get executed whenever the thread goes to *Alertable Wait State*. Threads that work with IO Manager and similar resources are found to be in alertable state frequently. But this schedule is something that we don't have direct control on. In

practice, we can safely assume it to happen randomly. We can have 3 kinds of continuity in the execution. They are illustrated in Figure3.1.

1. The succeeding emulator can execute without delay. (APC 2 starts right after APC 1)
2. The succeeding emulator can execute with a delay. (APC 5 is delayed after APC 4)
3. The succeeding emulator can start execution before completion of the current emulator. (APC 7 starts while APC 6 is in execution)

In case 1, we have a smooth continuity. In case 2, we need to wait for some time. This makes the execution slower but doesn't hurt us. In fact, these delays work as random sleeps and add to our stealth. Executions of 3rd kind aren't quite useful for single threaded malware. We can safely skip them. But they can also be used for system maintenance. They can queue more emulators in the system and manage resources to help coordination.

DIME doesn't create/own any thread. Thus we can't regulate the APC consumption by a thread. But a higher rate of emulator execution can be realized by infecting more threads. This way, we can control the rate at which emulators get executed.

DIME can be broken down to two phases:

1. Offline Processing
2. Online Processing

Offline Processing

During offline processing, we will take the original malware binary and split it into pieces called chunks. Emulators will be executing these blocks in the required order to perform the malicious action. We have used the splitting mechanism described in [16]. They have provides 3 ways of splitting the binary:

1. Basic Block Splitting (BBS)
2. Below AV Signature Threshold (BAST)
3. Paranoid mode

BBS splits the binary at the basic block level. BAST mode will split even the basic blocks according to a configurable threshold. The Paranoid mode chops the binary such that each block will only have a single instruction. The splitting will add necessary metadata and a few instructions to the actual set of instructions. Metadata caries the information required to execute them in the context of a victim thread. BBS results in the largest sized chunks while Paranoid mode results in the smallest of them. It should be noted that the smaller chunks improve our stealth. But they also increase the overhead of execution. In practice, the least stealthy splitting mode, BBS is found to be sufficient to evade anti-virus detection.

Online Processing

DIME is an anti-virus evasion framework. Alike [22] [16] [21] [17] it does not concern itself with the exploitation. The exploit used shall vary with the system. Usually, it depends on the OS version, applications installed, etc... We assume that the attacker has an exploit which can run the first emulator. All of the online processing of DIME happens in emulators.

The first emulator to get executed will do the necessary initialization. Initialization can also be split and executed by multiple emulators for extra stealth. Initialization includes setting up communication channels, identifying victim processes, etc... Once the initialization is completed, emulators will start executing chunks. After the execution of a chunk, the emulator will broadcast the state information and the next chunk to execute. The emulator that spawns next will use this data to execute the next chunk. The sequence of chunk execution is dynamic. For example, the chunk to execute after a conditional statement can only be determined at run-time. The chunks are appended with necessary instructions to handle these complications. A chunk execution will leave the id of the next chunk to execute in `ebx` register.

Multi-threaded malware programs are realized by allowing multiple emulators to execute blocks simultaneously. The maximum number of emulators that execute blocks concurrently is equal to the number of threads in the original malware.

3.0.2 Emulator

Executing code in the context of an alien thread is not easy. Execution in multiple of such threads owned by distinct processes, yet with coordination is harder. We developed an environment virtualizer, called emulator to help with this execution.

The emulator can get injected to any thread of any processes. This requires the emulator code to be position independent [27]. An emulator execution will only execute one chunk. They die out after the short execution as APC and respawns later in possibly a different thread. Thus they won't have persistent storage of their own. They will not have any knowledge of variables or memory regions used by their predecessors. A robust inter-emulator communication channel needs to be established to solve this. The challenge is even bigger since DIME allows delay between emulator executions. Thus the communication channel should be persistent even if there are no emulators running. Windows provides a variety of APIs for inter-process communication. But they are all under possible surveillance of malware scanners. Thus repeated calls to these APIs may lead to detection. We need a covert channel that can broadcast the information to emulators to come in the future. Most of the existing covert channels require both sender and receiver to be active at the same time. But this is not guaranteed in DIME. Section 3 point number 2 (page 16) is an example. Thus we discovered a new covert channel that suits our need.

3.0.3 SCBC (Semaphore based Covert Broadcasting Channel)

We have the following scenario:

1. A sender thread, S_1, who wants to broadcast to receiver threads R_1, R_2, R_3, ... (S_1 can also take the role of a receiver after initiating the broadcast.)
2. The broadcast should be persistent even after the exit of S_1.
   i.e. R_i may start after the exit of S_1.
3. The communication should be covert.

We invented SCBC, a new robust covert broadcasting channel using semaphores that satisfy all the above conditions. In a generic setup, SCBC also has the following advantages over the existing covert channels:

1. The integrity of data transferred is as good as the robustness of OS.
2. There can be multiple listeners.
3. Multiple senders can use the same channel in a time sliced manner.
4. The data in the channel persist even after the exit of the sender.

Overview

While DIME is a windows specific framework, SCBC is more generic. It works in any generic OS including Windows and Linux based operating systems. The only requirement is the availability and accessibility of semaphores. The idea of SCBC is to use the semaphore counter as the storage medium. The information will be transmitted as integers. The sender can create a semaphore and set the counter to the integer data that need to be communicated. The counter can be set during creation or by repeated calls to release or wait APIs.

Once the semaphore is created, listeners can open the semaphore object and retrieve the counter value to get the data.

Implementation

SCBC has a very simple design as well as implementation. Linux has two kinds of semaphores, *POSIX Semaphore* and *System V Semaphores* [19].

- *System V Semaphores:* This kind of semaphores are created in sets. `semget()` function can be used to create a set of System V Semaphores. The ability to create an array of semaphores positively affect SCBC by increasing our bandwidth. `semctl()` function can be used to perform actions on semaphore [1]. `semctl()` along with `GETVAL` flag can be used to get the value of the semaphore counter. `semop()` is the function used to acquire or release the ‘semaphore.
- *POSIX Semaphore:* POSIX Semaphores can be named or unnamed. We are more interested in named semaphores. `sem_open()` function can be used to create a named POSIX semaphore. `sem_post()` and `sem_wait()` functions respectively increment and decrement a semaphore's value. The value of semaphore-counter can be retrieved using `sem_getvalue()` function.

It can be seen that the Linux-Semaphore values can also be retrieved by direct API calls. The procedure gets a little complicated in Windows. `CreateSemaphore` API can be used to create a semaphore with a given name [14]. `ReleaseSemaphore` is used to increment the semaphore counter `WaitForSingleObject` or `WaitForMultipleObjects` [1] APIs can be used to decrement the semaphore-counter(s). Windows don't provide direct API(s) for retrieving the value of semaphore-counter. But `ReleaseSemaphore` API provides the previous count of semaphore object after incriminating the same. We call wait API first. Then we call the `ReleaseSemaphore` API to retrieve the previous count. One added to the previous count is the broadcasted number. The wait API call prior to the release API balances the counter increment done by the latter. But this is not sufficient while we have multiple listeners. More than one listener could call the wait function before any of them call the release function. This affects the integrity of transmission. But the issue can be easily solved by associating a mutex for the semaphore. Every listener will have to first lock the mutex to read the value.

SCBC reduces a semaphore to a simple shared integer variable. Applying the same concepts to mutexes, one can build shared boolean variables. Any protocol that shall be implemented using shared variables can be implemented using SCBC. A simple example is to use two SCBC semaphores to broadcast a stream of integers. One semaphore can be used for data and one for the index.

3.0.4 DIME Communication Mechanisms

DIME has a very flexible architecture. Its inter-emulator communications can be modified to fit the preferred mode of communication. To avoid detection, it is best not to follow a recognizable pattern. One may use different kinds of covert channels as well as inter-process communication mechanisms. Changing the channels dynamically at run-time will further complicate detection.

We will discuss a design using two types of communication channels. The architecture is illustrated in Figure 3.2. We have a *Primary* and a *Secondary* Channel. Primary

[1] You may have multiple semaphores to broadcast multiple integers simultaneously.

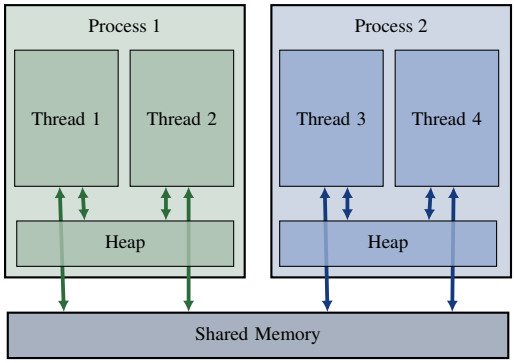


Figure 3.2: DIME Communication Channels - Design 1
Shared memory is used as the primary channel and heap memory is used as the secondary channel. The primary channel is unique for the system while each victim process will have its own secondary channel.

Channel will be unique for the system and accessible to all the emulators. Secondary Channel will be unique for a (an infected) process. Its access will be restricted to threads of a specific process. All the process-specific information can be stored here. These channels shall be built using Windows provided data sharing mechanisms. We discuss an approach using Heap and Shared memory under Section 4.0.2. The role of SCBC here is to nullify the initialization requirements of these channels.

A slight modification to this design can be obtained by adding multiple primary channels. This is illustrated in figure 3.3. Here *Process 1* and *Process 2* shared a primary channel (*Shared Memory 1*). *Process 2* and *Process 3* shares another primary channel (*Shared Memory 2*). Emulators running in process 2 shall take the responsibility of synchronizing both primary channels. This design reduces the dependency on a particular primary channel. Such a design improves the distributed and decentralized nature of the architecture. It improves the resilience of the system greatly. With multiple primary channels, any destruction or loss of a primary channel will not stop the execution. The lost channel shall be rebuilt using the existing channels. Note that, this design is effective to counter any attempt of removing DIME from the system by the defender.

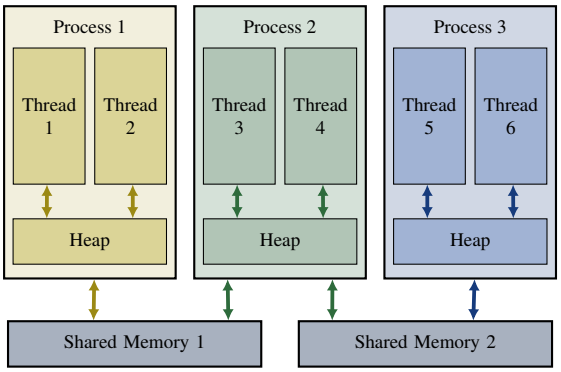


Figure 3.3: DIME Communication Channels - Design 2
Shared memory is used as the primary channel and heap memory is used as the secondary channel. A set of processes will share the primary channel while each victim process will have its own secondary channel.

CHAPTER 4

Implementation

DIME is the first attempt to distribute a malware or program execution without creating or owning a thread. The executions happen in the context of foreign or victim threads. Executing the chunks in the foreign context without errors is challenging. The coordination of chunk executions is another challenging aspect of DIME. In this chapter, we will discuss the most important implementation aspects of DIME.

4.0.1 Emulator

Emulator is the function to be executed as APC. It should be a single function with the signature of `PAPCFUNC` [2]. Since the function is to get executed in the context of a thread under a foreign process, the code should be position independent [27]. Our implementation of emulator is a mix of assembly as well as C++ code.

4.0.2 Communication Channels

We observe that the *Heap Memory* allocated or *Shared memory* created/attached by a thread is also accessible to every other thread of the process. The memory mapping will remain intact until the process de-allocates or detaches it explicitly or till the end of the process. These observations recommend shared memory as a good candidate for the primary channel. Since heap is local to a process, it is a good choice for the secondary channel. The catch is that only the thread that allocated/created/attached the heap/shared memory will know of the range of virtual address space allocated. Thus the starting address of the allocated space needs to be explicitly communicated to emulators to come in future. One way out is to attempt to create/attach the shared memory at a pre-defined virtual address range. But this is not fail-safe. The requested address could be already under use! In such scenarios, the memory allocation APIs will use

an address range of their choice. We take the help of SCBC here. As soon as an emulator creates/attaches the primary channel, it will broadcast the starting address of the memory region using SCBC. The starting address of the secondary channel can be broadcasted throught the primary channel. The emulators to come in future shall get the address range from SCBC and use it without calling create/attach APIs.

4.0.3 Challenges of Distribution

Distributing the malware execution creates a number of implementation challenges. Thread local and process local resources now have to be shared among victim processes. The process specific information like process-id and executable name need to be virtualized. We will discuss the major challenges here.

Implementing Stack and Heap

Stack is a thread local resource and heap is a process local resource. All the emulators should be able to access them. Thus, these resources need to be shared with all the victim processes. This can be implemented in multiple ways. We followed an approach similar to [16]. We allocate one shared memory region per thread of the original malware and use it as stack. The emulator will set `esp`, `ebp` as well as other registers to point to appropriate locations in this region just before chunk execution. Similarly, we allocate a shared memory region for heap. Custom heap management APIs are also provided to give it the effect of a normal heap.

Handling files and sockets

A file or a socket descriptor created by a thread will be accessible to the specific process only. This poses a challenge. DIME could be opening a file from one process and writing to it from a different process. Similarly for sockets. This complicates the use of functions like `fopen()` and `fprintf()`. Nevertheless, Windows provides an easy way of handling this problem. Windows provides its own APIs to handle files and sockets. Unlike standard C libraries, the programming interface for these APIs are

Handles.

A Handle under the hood is only a `void*` [15] [1]. But they are well managed by windows. Handles can be duplicated and shared among multiple processes. As soon as an emulator creates a handle, it duplicates the handle for all the infected processes. The information is then broadcasted over primary channel. Emulators spawn in other processes after the broadcast will use the duplicated handles. Custom APIs shall be provided for file and socket handling to enable the use of standard C functions as well. These custom APIs can be simple wrappers for Windows APIs.

Process-specific functions

The functions whose behavior is strictly dependent on the process are called process specific functions. `GetCurrentProcessId()` and `GetCurrentDirectory()` are examples of such functions. DIME does not have a process of its own. It is a completely decentralized system. There is no master or core process. Thus process specific functions cannot be used directly. Special APIs are provided to handle these functions.

[1] `HADNLE` is defined as `typedef PVOID HANDLE;` and `PVOID` is defined as `typedef void *PVOID;`

CHAPTER 5

Evaluation and Results

In this chapter, we discuss the results of our experiments with our PoC [1]. We selected five verities of malwares for the test. The samples were selected to cover the most common functionalities and API calls made by malware in the wild. No care is taken to ensure the stealth of the malware.

- **Offline Keylogger:** This malware records the keystrokes typed by the user. The data is written to a file in `%TMP%` directory. This is one of the most common malware. This malware repeatedly calls `GetAsyncKeyState` API to track keystrokes. The repeated API calls contribute to a definitive behavioural pattern. Thus these malwares are quite vulnerable to behavioral analysis.

- **Remote Keylogger:** Remote keylogger tracks and sends the keystrokes of user to the attacker over the network. A data packet is sent for every key that is pressed. Frequent network usage along with repeated calls to the sensitive APIs makes this malware susceptible to behavioral analysis. This malware also edits windows registry. Registry edit being very sensitive action in Windows, the malware is unconcealed.

- **Backdoor:** This is a backdoor written in C++. The malware enables the attacker to execute custom commands on the victim system. This malware can get easily flagged due to its network usage as well as the usage of critical APIs like `ShellExecute()`, `NtShutdownSystem()`, etc... Since these malwares are quite common, the signature will be readily available in the database.

- **Ransomware:** Ransomware is very dangerous. They are built on pure monetary interests. This malware encrypts files in the system. In the wild, attackers demand money for providing the encryption key. The files can be decrypted to retrieve data only with the encryption key. File indexing, as well as encryption operations, are the detectable behaviors of this malware.

[1] Refer to Appendix section A to find the link to our PoC.

- **Screenshot malware:** This malware takes the screenshot of the system every second. The files are stored in `%TMP%` directory. The repeated calls to APIs like `CreateCompatibleBitmap()`, which are used to monitor resources/devices connected to the system is a detectable signature.

The above malwares cover the majority of the common aspects of malwares. This includes system monitoring, network activities, file handling, file indexing, encryption and resource/device monitoring.

All the malwares were written in C++ and obfuscated with DIME. The tests were conducted in 64 bit Windows 10. Since the PoC of emulator is written for 32 bit architecture, 32 bit programs were used as targets. We evaluated DIME by targeting some of the most common software such as Chrome, Opera, VLC, Acrobat Reader, etc...

5.0.1 Detectability

To test detectability, we selected 10 top rated anti-viruses. DIME-obfuscated malwares were run on the system with the anti-virus. The results of the test are given in Table 5.1. Table 5.1 remains the same for a varying number of victim processes as well as different splitting mechanisms. None of the anti-viruses was able to detect DIME. We like to note here that Norton and Bitdefender were able to detect the injection of DIME. But DIME, being an execution framework does not worry about the injection. It is assumed to be achieved by the attacker via exploits.

5.1 Performance

Analyzing performance counters for anomalies is an emerging technique to detect malware [10] [29]. We injected DIME to selected processes and measured the performance impacts on the processes. Resulted CPU usages by processes are given in figure 5.1. The graphs show the CPU usage of a process during the start, normal usage and when infected by DIME.

Though the CPU usage spikes to a considerable amount when injected to a single process, the load is reduced as the number of infected processes increase. When the

Anti-virus \ Malware	Remote Key-logger	Remote Key-logger	Backdoor	Ransomeware	Screenshot Malware
BitDefender	×	×	×	×	×
Norton	×	×	×	×	×
Kaspersky	×	×	×	×	×
WEBROOT	×	×	×	×	×
McAfee	×	×	×	×	×
ESET	×	×	×	×	×
Avast	×	×	×	×	×
AVG	×	×	×	×	×
Windows Defender	×	×	×	×	×
Avira	×	×	×	×	×

Table 5.1: Detectability of DIME obfuscated malware.

✓ Detected

× Undetected

The table remains the same for a varying number of victim processes or threads and different splitting mechanisms.

number of infected processes is three or more, the CPU usage spikes are affordable. It should be noted that the victim processes continued to serve their benign purpose under infection. During all tests, victim processes ran without any user-noticeable performance degradation.

Figure 5.2 shows the values of major performance counters. The test was conducted by injecting to only chrome. Injection to single process is expected to show maximum variation in the counters. Yet, no noticeable variation is seen.

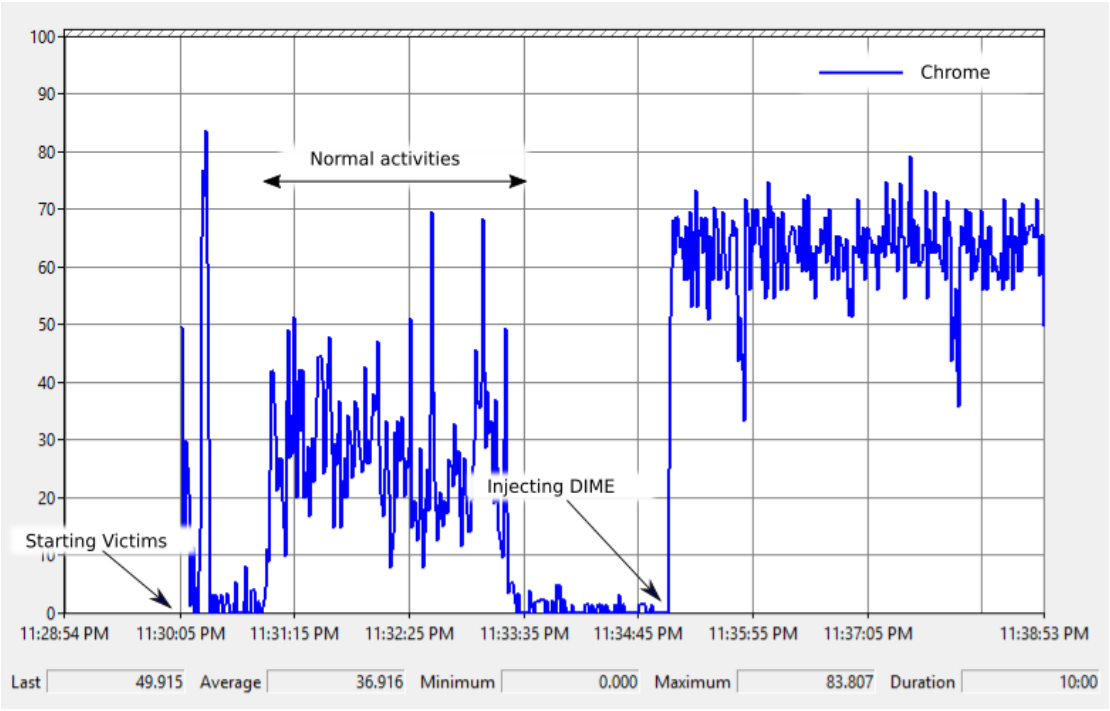
5.1.1 Effect of different chopping mechanisms

The PoC works with different chopping mechanisms described in section 3.0.1. None of the anti-viruses in Table 5.1 is able to detect any of the chopping modes. In the paranoid mode, malware execution is found to be slow. The key-logger is found to miss certain keys as compared to BBS mode.

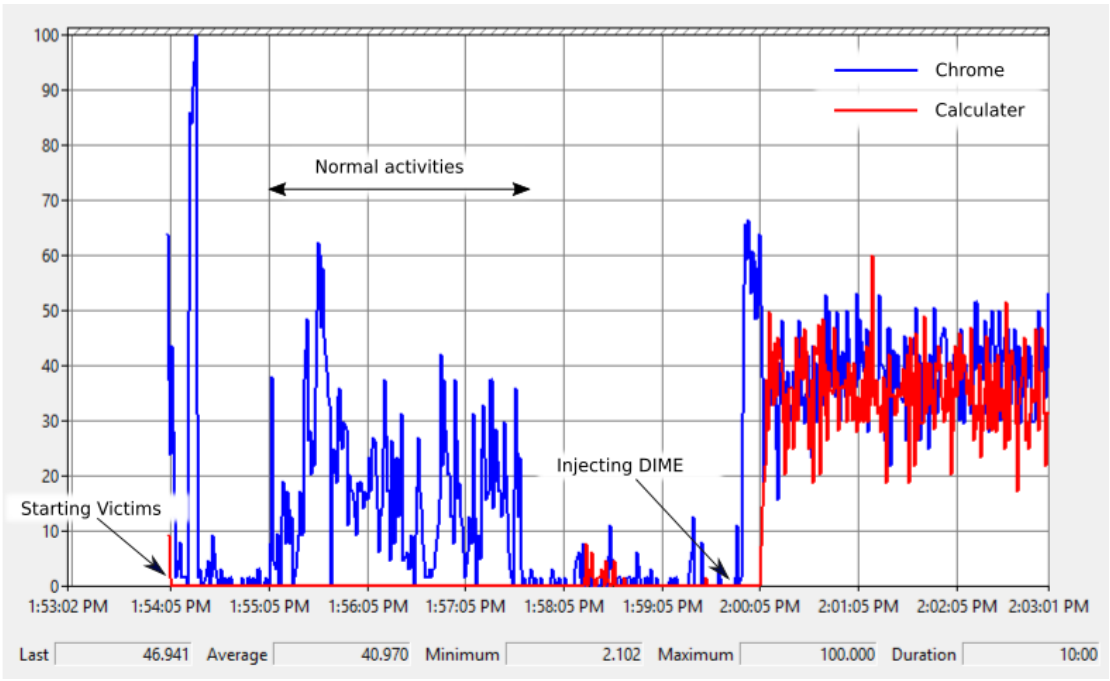
5.1.2 Effect of victim process exit

When the window of an infected process is closed, though the window closes, DIME continues to work. The threads do not exit until all the queued APC functions (emulators) are emptied. By fast queuing of emulator in non-graphical threads, the process continues to live in the background and serves as a host for DIME.

The test results prove that DIME is a practical obfuscation technique to avoid behavioral and heuristic detection.

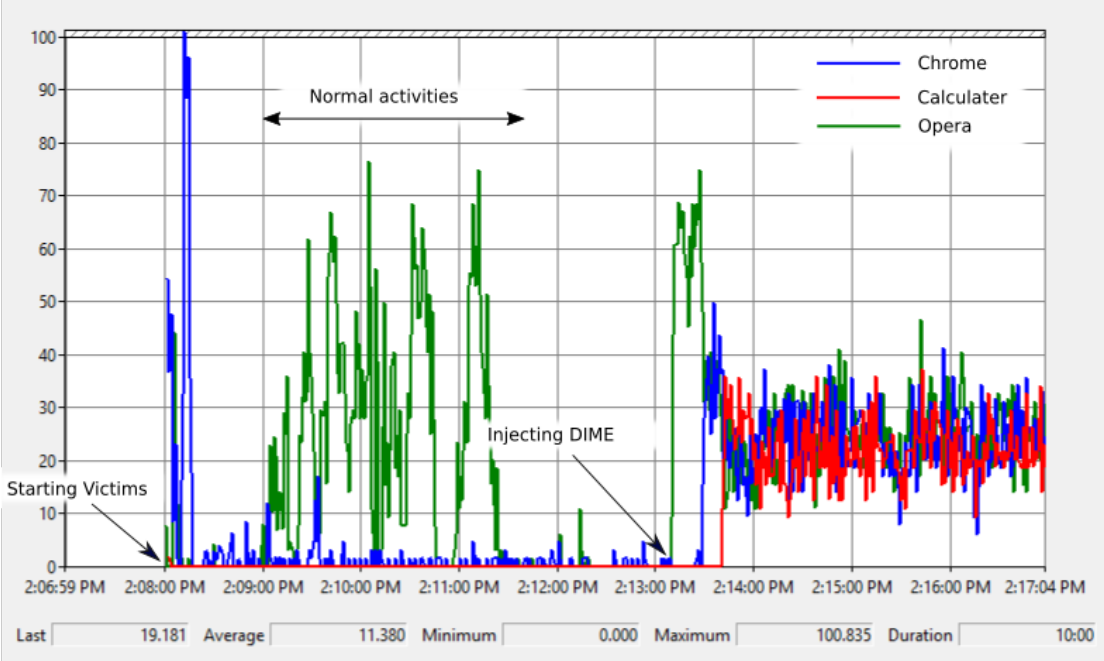


(a) DIME injected to Chrome.
Normal activities represent a typical browsing activity in Chrome.

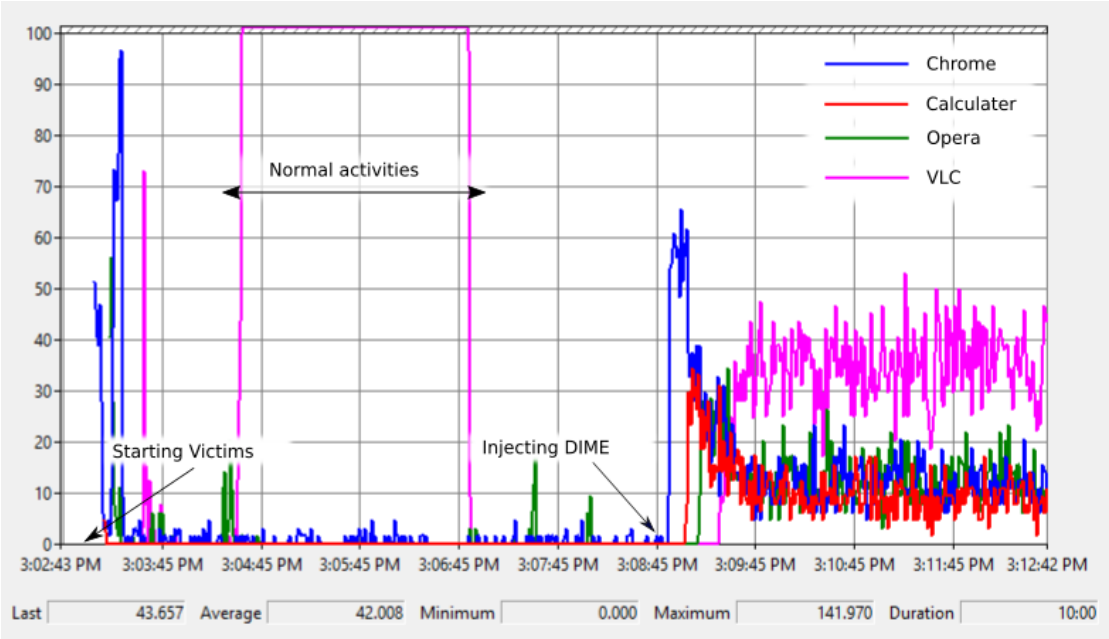


(b) DIME injected to Chrome and Calculator.
Normal activities represent a typical browsing activity in Chrome.

Figure 5.1: CPU usage vs Time graph.
Performance impact of DIME with varying number of infected processes.

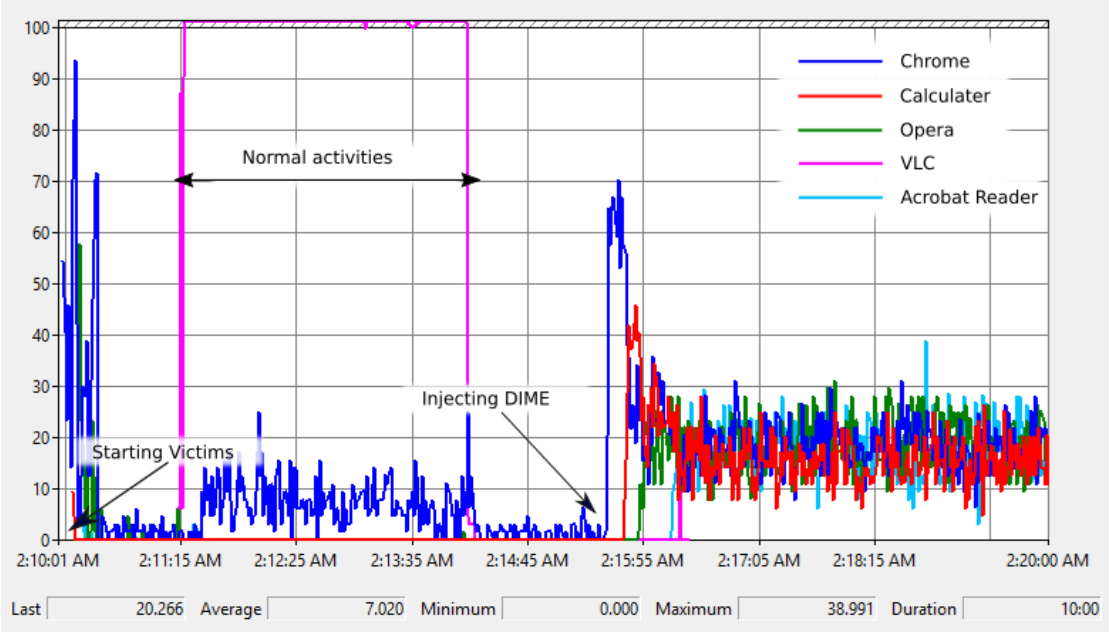


(c) DIME injected to Chrome, Calculator, and Opera.
Normal activities represent a typical browsing activity in Opera.



(d) DIME injected to Chrome, Calculator, Opera, and VLC media player.
Normal activities represent a video being played in VLC media player.

Figure 5.1: (cont.) CPU usage vs Time graph.
Performance impact of DIME with varying number of infected processes.



(e) DIME injected to Chrome, Calculator, Opera, VLC media player, and Acrobat Reader. Normal activities represent a media being played in VLC and a typical web surfing performed on Chrome.

Figure 5.1: (cont.) CPU usage vs Time graph.
Performance impact of DIME with varying number of infected processes. The performance impact is found to decrease with the increase in the number of infected processes. The infected processes continue to work without a user-observable performance impact.

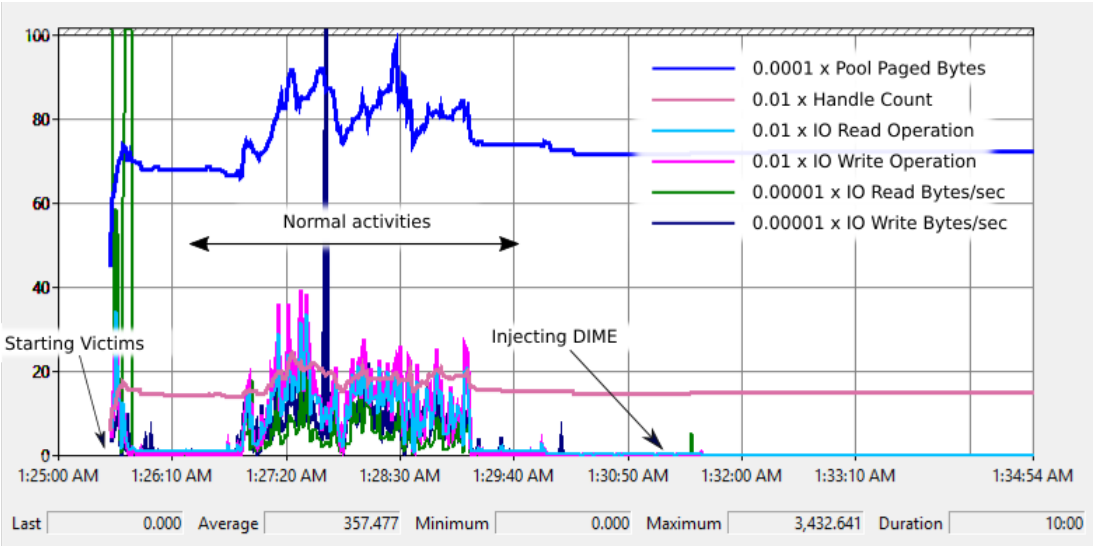


Figure 5.2: Performance counters of Chrome with DIME infection.
DIME was injected to only chrome to maximize anomalies in the counter values. Note that no noticeable variation is found.

CHAPTER 6

Countermeasures

Section 5 has shown that DIME is an effective way of hiding malware. This is very alarming. Countermeasures need to be taken to prevent the attackers from using this technique to harm computer systems. In this chapter, we suggest some of the countermeasures that can be taken to reduce the damage caused by DIME.

6.1 Prevention

DIME is built upon a core windows feature called APC. We find that the APCs are tightly coupled with the way the threads are dispatched. Also, almost all software use APCs extensively to improve their performance. Thus removing the APC feature is not practical. Nevertheless, the APC feature is inherently insecure and needs strict restrictions. We suggest the following countermeasures to secure systems against DIME:

1. *Queue Privilege Class*: A new privilege class shall be introduced for processes with the permission to queue APCs. This class shall be thread specific. System processes can be added to the circle by default. The system processes are added to the list under the assumption that they are not compromised. If these processes with elevated privileges are compromised, defensive actions are limited. When a process creates or forks a new process, an optional flag can be provided to add the new process to Queue Privilege Class. Optionally a process can add more threads/processes to its Queue Privilege Class. This will prevent malign processes/threads from queuing APCs on their targets.

2. *Limit Recursive Queuing*: The legitimate needs for recursive queuing of APCs is very narrow. DIME's continuity is ensured by the recursive queuing of APCs. A maximum depth shall be introduced for recursive queuing of APCs. This will prevent the self-regeneration of emulators.

3. *Restrictions on handle duplication*: Duplication and sharing of handles is a crucial trick used by DIME in its implementation. Duplication, as well as sharing of

handles, across processes can be restricted. Most of the legitimate requirements of handle sharing can be achieved by providing special APIs. These APIs will come at a minor performance impact. For example, file handles are usually shared to maintain consistency. Same writing location needs to be maintained if multiple processes are writing to the same file. An explicit call to `SetFilePointer()` can solve this problem.

4. *Disable APC Queues*: It is not practical to disable APC Queues in Windows. But the feature can be disabled for threads which do not explicitly use it.

5. *Eliminating SCBC*: Providing minimal information to the client is a powerful approach to secure systems. SCBC is a result of the negligence of this approach. Conceptually, semaphores are used to limit the number of users of a resource. The knowledge of the number of current users is not a requirement of semaphore. Revealing the counter is clearly a design flaw in the OS. Sadly, all the major operating systems have this flaw. SCBC can be prevented by not revealing the semaphore-counter value to clients.

6.2 Detection

Detecting DIME using behavioral and heuristic methods is intricate if not impractical. Such an analysis would require the anti-virus to find connections between DIME executions across processes and threads. This is a hard problem. DIME executions, which are emulators do not carry any common identifiers. They are small execution events occurring benign threads randomly.

A possible way of detecting DIME is by detecting its communications channels. Robust communication channels are necessary for DIME to function properly. Detecting shared memory regions and other modes of communications can be a detection strategy. Note that DIME may use any kind channel for the inter-emulator communication. It can also change the communications channels dynamically. Detection of side-channels that are dynamically changing is a challenging problem.

A practical approach of detecting DIME would be by tracking the CPU usage by processes. Practically, only a limited number of processes will have a considerably high CPU usage in OS at a time. We notice that almost all the DIME infected processes

have similar CPU usage. The pattern of CPU usage by multiple processes can be used to detect the presence of DIME. However, by varying the number of infected threads in different processes DIME can manipulate the CPU usages. Infecting a large number of processes will reduce the CPU usage considerably. These methods will make the above-mentioned detection method challenging.

6.3 Removal

Removing DIME from an infected system is quite complicated. The defender should remove all the emulators simultaneously. Given the APC implementation in Windows, this is not practical. Another way of removing DIME is to destroy the inter-emulator communication channels. Without proper inter-emulator communications, DIME cannot work. Destroying the channels can get really complicated. Similar to the emulator removal, all the channels should be destroyed simultaneously. Emulators can use any channel left in the system to bring back other channels.

CHAPTER 7

Future Works

DIME is a highly scalable framework. It has a very flexible architecture. The emulator, as well as the communication channels, can be modified to suit a particular system.

An efficient implementation of emulator can enhance the stealth of DIME to a large extent. As a future work on DIME, we hope to improve our emulator implementation. Communication channels are another part of DIME that needs improvements. A design that requires minimal inter-emulator communications will result in the most stealthy execution of malware. Switching communication channels at run-time can add more randomness to the behavior of DIME. A dynamic channel switching protocol can be developed using selected inter-process communication mechanisms and side channels. These improvements can make DIME even more stealthy.

Currently, the emulator executions rates are purely depended on the victim processes. We control the frequency of execution by varying the number of infecting threads. Many times, emulators get spawn while another emulator is already executing a chunk. These emulators do only house-keeping tasks. This adds a lot of overhead and consumes CPU. A protocol needs to be developed to control emulator queuing depending on the characteristics of the target system. The protocol should minimize the number of emulator executions without compromising the resilience of the framework.

In this thesis, we have proposed SCBC, a new covert channel. SCBC reveals a common designs flow in Operating Systems. This is something that has a lot of scope for further study. A practical approach to designing systems providing minimal information to its clients needs to be explored.

With DIME, we have suggested a robust way of obfuscating the malware against behavioral and heuristic detection techniques. The most important future work on this is to find a way to detect DIME effectively. A new class of detection techniques needs to be invented to prevent distributed malware execution efficiently.

CHAPTER 8

Conclusion

In this work, we have suggested an effective obfuscation technique to avoid behavioral detection. None of the major anti-virus programs are able to detect DIME. Effective detection evasion was achieved by distributing malicious execution across threads of multiple benign processes. The success of DIME urges the invention of new classes of detection techniques.

We have also proposed a new covert channel for Operating Systems. SCBC reveals a common design flow in systems. One should always try to minimize the amount of information revealed from a system. Revealing only the essential information in general leads to more secure systems.

APPENDIX A

PROOF OF CONCEPT

Our working code of DIME is available at

`https://gitlab.iitm.ac.in/JithinPavithran/DDP`

REFERENCES

[1] (). URL http://man7.org/linux/man-pages/man2/semctl.2.html.

[2] (). Queueuserapc function. URL https://docs.microsoft.com/en-gb/windows/desktop/api/processthreadsapi/nf-processthreadsapi-queueuserapc.

[3] (1998). *J. Comput. Secur.*, **6**(3). ISSN 0926-227X.

[4] A virus detection scheme based on features of control flow graph. *In 2011 2nd International Conference on Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC)*. 2011.

[5] Frankenstein: Stitching malware from benign binaries. *In Presented as part of the 6th USENIX Workshop on Offensive Technologies*. USENIX, Bellevue, WA, 2012. URL https://www.usenix.org/conference/woot12/workshop-program/presentation/Mohan.

[6] **Bonfante, G.**, **M. Kaczmarek**, and **J.-Y. Marion**, Control flow graphs as malware signatures. *In International workshop on the Theory of Computer Viruses*. 2007.

[7] **Borello, J.-M.**, **É. Filiol**, and **L. Mé** (2010). From the design of a generic metamorphic engine to a black-box classification of antivirus detection techniques. *Journal in Computer Virology*, **6**(3), 277–287. ISSN 1772-9904. URL https://doi.org/10.1007/s11416-009-0136-2.

[8] **Bruschi, D.**, **L. Martignoni**, and **M. Monga**, Detecting self-mutating malware using control-flow graph matching. *In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2006.

[9] **Christodorescu, M.**, **S. Jha**, **D. Maughan**, **D. Song**, and **C. Wang**, *Malware detection*, volume 27. Springer Science & Business Media, 2007.

[10] **Demme, J.**, **M. Maycock**, **J. Schmitz**, **A. Tang**, **A. Waksman**, **S. Sethumadhavan**, and **S. Stolfo**, On the feasibility of online malware detection with performance counters. *In ACM SIGARCH Computer Architecture News*, volume 41. ACM, 2013.

[11] **Dolan, S.** (2013). mov is turing-complete. *Cl. Cam. Ac. Uk*, 1–4.

[12] **Eskandari, M.** and **S. Hashemi** (2011). Metamorphic malware detection using control flow graph mining. *Int. J. Comput. Sci. Network Secur*, **11**(12), 1–6.

[13] **Ferrie, P.** (2007). Attacks on more virtual machine emulators. *Symantec Technology Exchange*, **55**.

[14] **GrantMeStrength** (). Semaphore objects - windows applications. URL `https://docs.microsoft.com/en-us/windows/desktop/sync/semaphore-objects`.

[15] **GrantMeStrength** (). Windows data types - windows applications. URL `https://docs.microsoft.com/en-us/windows/desktop/winprog/windows-data-types`.

[16] **Ispoglou, K. K.** and **M. Payer**, malwash: Washing malware to evade dynamic analysis. *In 10th USENIX Workshop on Offensive Technologies (WOOT 16)*. USENIX Association, Austin, TX, 2016. URL `https://www.usenix.org/conference/woot16/workshop-program/presentation/ispoglou`.

[17] **Jana, S.** and **V. Shmatikov**, Abusing file processing in malware detectors for fun and profit. *In 2012 IEEE Symposium on Security and Privacy*. 2012. ISSN 2375-1207.

[18] **Jeong, K.** and **H. Lee**, Code graph for malware detection. *In 2008 International Conference on Information Networking*. 2008. ISSN 1550-445X.

[19] **Kerrisk, M.**, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, San Francisco, CA, USA, 2010, 1st edition. ISBN 1593272200, 9781593272203.

[20] **Lu, Y.-B.**, **S.-C. Din**, **C.-F. Zheng**, and **B.-J. Gao** (2010). Using multi-feature and classifier ensembles to improve malware detection. *Journal of CCIT*, **39**(2), 57–72.

[21] **Min, B.** and **V. Varadharajan**, Design and analysis of a new feature-distributed malware. *In 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. 2014. ISSN 2324-898X.

[22] **Oberheide, J.**, **M. Bailey**, and **F. Jahanian**, Polypack: an automated online packing service for optimal antivirus evasion. *In Proceedings of the 3rd USENIX conference on Offensive technologies*. USENIX Association, 2009.

[23] **Runwal, N.**, **R. M. Low**, and **M. Stamp** (2012). Opcode graph similarity and metamorphic detection. *Journal in Computer Virology*, **8**(1-2), 37–52.

[24] **Santos, I.**, **F. Brezo**, **X. Ugarte-Pedrero**, and **P. G. Bringas** (2013). Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, **231**, 64–82.

[25] **Shabtai, A.**, **R. Moskovitch**, **C. Feher**, **S. Dolev**, and **Y. Elovici** (2012). Detecting unknown malicious code by applying classification techniques on opcode patterns. *Security Informatics*, **1**(1), 1.

[26] **Shacham, H.** *et al.*, The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). *In ACM conference on Computer and communications security*. New York,, 2007.

[27] **Sikorski, M.** and **A. Honig**, *Practical malware analysis*. No Starch Press,us, 2012.

[28] **Spinellis, D.** (2003). Reliable identification of bounded-length viruses is np-complete. *IEEE Transactions on Information Theory*, **49**(1), 280–284. ISSN 0018-9448.

[29] **Tang, A.**, **S. Sethumadhavan**, and **S. J. Stolfo**, Unsupervised anomaly-based malware detection using hardware features. *In International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014.

[30] **Ye, Y.**, **D. Wang**, **T. Li**, and **D. Ye**, Imds: Intelligent malware detection system. 2007.

[31] **You, I.** and **K. Yim**, Malware obfuscation techniques: A brief survey. *In 2010 International conference on broadband, wireless computing, communication and applications*. IEEE, 2010.