©Copyright 2021
Kalana Sahabandu

RESTful and Light Weight Dynamic Information Flow Tracking-Based Computer Security Systems

Kalana Sahabandu

A thesis
submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

University of Washington

2021

Committee:

Radha Poovendran, Chair

Payman Arabshahi

Program Authorized to Offer Degree:

Electrical and Computer Engineering

University of Washington

Abstract

RESTful and Light Weight Dynamic Information Flow Tracking-Based Computer Security Systems

Kalana Sahabandu

Chair of the Supervisory Committee:
Professor Radha Poovendran
Department of Electrical and Computer Engineering

Cyber threats impose a significant service disruption to public and private sectors at an alarming rate. Modern cyber threats such as ransomware and advanced persistent threats are intelligent and stealthy. These threats are able to successfully avoid detection from traditional cyber defenses such as static signature based anomaly detectors and anti-virus software. However, the activities of the cyber adversaries introduce information flows in the victim system that capture the data and control commands used by the cyber adversaries.

Dynamic Information Flow Tracking (DIFT) is a promising cyber defense mechanism that tags suspicious information flows, tracks their propagation in the system and performs security checks to validate the authenticity of the information flows to detect cyber threats [17, 29]. LIBDFT [17] is a widely used DIFT implementation in systems security research. However, current version of LIBDFT is not compatible with the Intel Pin, a dynamic binary instrumentation framework that supports LIBDFT code base, used by the most recent Linux distributions. LIBDFT also does not scale to perform DIFT on multiple programs. Moreover, it does not provide an ability to remotely start/terminate DIFT on a host system.

This thesis is organized as follows. The Chapter one provides introduction to the problem analyzed in this thesis, proposing DIFT implementations that allows deployment of DIFT-

based detection mechanisms to consumer space. The chapter two provides the necessary preliminaries on DIFT functionality, LIBDFT architecture and the limitations of LIBDFT that restricts its deployment in modern Linux distributions. The Chapter three of this dissertation introduces RESTful DIFT that updates the code base of LIBDFT library to support the latest version of Intel Pin tool and enables deployment of DIFT in modern Linux distributions. Also RESTful DIFT enables multi program DIFT functionality to improve the scalability of LIBDFT library. Moreover, it adds a RESTful service layer for improving the usability of LIBDFT. A real world attack example, Screengrab attack is used to validate the functionality of RESTful DIFT implementation. The Chapter four of this dissertation introduces Light Weight DIFT, a DIFT architecture with low memory and run-time overhead. It also supports performing DIFT on multi process programs. Moreover, Light Weight DIFT provides a user friendly graphical visualization of the DIFT results. The validity of the Light Weight DIFT implementation is verified using multi-host Screengrab attack. Lastly, the Chapter five provides a discussion on future directions that can further improve the compatibly and scalability of RESTful DIFT and Light Weight DIFT, and concluding remarks.

TABLE OF CONTENTS

Page

List of Figures iii

Glossary v

Chapter 1: Introduction 1

Chapter 2: Preliminaries 4
- 2.1 Dynamic Information Flow Tracking (DIFT) 4
- 2.2 Intel Pin 6
- 2.3 LIBDFT 8
- 2.4 Limitations of LIBDFT 13

Chapter 3: RESTful DIFT 16
- 3.1 REpresentational State Transfer (REST) architecture 17
- 3.2 Process notifications through Netlink API 19
- 3.3 RESTful DIFT architecture 20
- 3.4 Analysis of Screengrab attack using RESTful DIFT 26
- 3.5 Limitations 29

Chapter 4: Light Weight DIFT 35
- 4.1 Neo4j 36
- 4.2 View from the victim's perspective Light Weight DIFT 36
- 4.3 View from the adversary's perspective Light Weight DIFT 38
- 4.4 Analysis of Multi-host Screengrab attack using Light Weight DIFT 39

Chapter 5: Future work and Conclusions 52
- 5.1 RESTful DIFT future work 52
- 5.2 Light Weight DIFT future work 54

5.3 Conclusion 55

Bibliography 57

Appendix A: Source code of the Screengrab 60

LIST OF FIGURES

Figure Number Page

2.1 Conventional vs Static Information Flow Tracking (modified) compiler 6
2.2 LIBDFT architecture [17] 9
3.1 RESTful architecture 19
3.2 Netlink header, kernel connector and process event connector structures . . . 21
3.3 Architecture of the RESTful DIFT 22
3.4 Database collections and relationships between them. 26
3.5 User dasboard of the RESTful DIFT's RESTful service component 27
3.6 RESTful DIFT prompt to grant user's permission for further execution of the program 30
4.1 Virtual network created within VirtualBox environment connecting adversary and victim computers 41
4.2 Screenshot file produced during the screengrab attack and the log file produced by the Netlink based logger 42
4.3 Complete process trace graph produced by the view from victim's perspective Light Weight DIFT's visualization tool 47
4.4 SCP process (Yellow node) created by the Linux background process (Blue node) during the execution of the multi-host screengrab attack 47
4.5 SSH process spawned by the Linux operating system (Red node) during the first SSH session from the adversary and it's children (Green nodes) created to execute commands required for the initial set-up of the attack 48
4.6 SSH process spawned by the Linux operating system (Red node) during the second SSH session from the adversary and it's children (Green nodes) created to carry out the attack 49
4.7 Background processes (Blue nodes) recorded during the time frame of the multi-host screengrab attack 50
4.8 Complete process trace graph produced by the view from adversary's perspective Light Weight DIFT's visualization tool. 51

5.1 Machine learning framework for automating cyber threat detection using Light Weight DIFT. 56

GLOSSARY

IA-32: is an abbreviation of Intel Architecture 32-bit, also commonly known as 32-bit architecture. IA-32 instruction set are 32-bits in size and the maximum amount of physical memory accessible by IA-32 system is 4 gigabytes.

X86-64: is commonly known as 64-bit architecture developed by AMD. x86-64 instruction sets are 64-bits in size and the maximum amount of memory accessible by x86-64 system is 2 terabytes.

MIC: stands for Many Integrated Core Architecture. MIC was originally developed by Intel which contains series of micro-architectures that integrates many physical cores into a single integrated circuit.

API: is an abbreviation of Application Programming Interface. APIs are utilized to create a connection between two computers or two computer programs. An API simplifies the programming by abstracting the under the hood implementation and only exposing the data or actions needed for the developers.

JIT: or Just-in-Time Compiler is a program that turns source code into instruction that can be sent to the computers processor. Unlike conventional compiler that turns source code into instructions before a program starts, Just-in-Time compilers compile the code during the run-time which is also known as on the fly compilation. Java and `C#` programming languages are some of the languages that uses this compilation technique.

CPU THREADS: are virtual components or cores that divides the physical CPU into virtual multiple cores. A single core CPU can have up to 2 threads. The threads are always created by the operating system to perform to tasks of an application. Threads allows CPUs to perform multiple tasks at the same time which makes tasks faster and more efficient.

PAGE TABLE: is a data structure utilized by the virtual memory system in a operating system to create a mapping between virtual memory addresses and physical memory addresses. Virtual addresses are typically used by programs executed by a process while physical addresses are used by the hardware devices connected to a computer.

Mapping between virtual addresses and hardware addresses are used by the operating systems for security purposes such that malicious programs does not have the ability to access actual physical memory.

MEMORY PAGE: also known as a page or a virtual page, is a fixed-size contiguous chunk of virtual memory, mapped to a single entry in the page table. It is the smallest amount of virtual memory that can be addressed by the operating system.

VIRTUAL MACHINE/COMPUTER: is a virtual environment that acts as a fully functional computer within a physical computer. A virtual machine runs on an isolated environment from its host computer with its own resources such as CPU, memory, network interface, an operating system and other resources.

SOFTWARE FRAMEWORK: is an abstraction that provides general functionality which can be changed by additional user written source code. A framework provides a standard way to build and deploy software that are universal and reusable. A software framework could have compilers, libraries, support programs and APIs.

HTTP: or Hyper Text Transfer Protocol is a client-server protocol that allows fetching resources such as HTML files, audio files, image files, etc. HTTP protocol is the foundation of any data exchange that happens in the web.

JSON: stands for JavaScript Object Notation is a lightweight data management format for storing and transporting data between programs or computer systems. JSON is widely when data is sent from a server to a client. JSON is human readable and easy to understand.

SYSTEM CALL: is a way that a program can request for a kernel service (e.g., fork, execv and dup)

ACKNOWLEDGMENTS

First and foremost I am extremely thankful to my advisors Prof. Radha Poovendran and Prof. Linda Bushnell for their indispensable advice, endless support and patience during my masters study. Their cosmic knowledge and abundant experience in academia as well as in industry have uplift me all the time during my research and daily life. I would also like to thank Prof. Payman Arabshahi for his acceptance to be in my masters dissertation committee.

I would like to thank Dr. Daniel Koller, program manager of Office of Naval Research (ONR) Cyber Program for his valuable comments during the program review. I thank Dr. Satish Chikkagoudar from Naval Research Laboratory (NRL), Center for High Assurance Computer Systems for interacting through multiple meetings over the course of many months to provide invaluable comments and feedback on the RESTful DIFT and Light Weight DIFT implementations to successfully complete the transition to NRL. I would also like to acknowledge the support of ONR MURI grant N00014-16-1-2710 P00002.

Lastly I thank all the members in Network Security Lab at University of Washington. It it their boundless help and support that made my academic journey at Network Security Lab and at University of Washington a magnificent time. Moreover, I thank Joey Allen from Georgia Institute of Technology for helping me by answering many questions during this work and for providing me with much needed links and support. Without everyone's monumental encouragement and support during past few years, it would be impossible for me to complete my masters journey.

DEDICATION

I dedicate my dissertation work to my family, Prof. Poovendran and many friends. A special feeling of gratitude to my loving parents, Udaya Sahabandu and Lalani Liyanage whose words of endless encouragement and push for tenacity ring in my ears. My brother Dinuka Sahabandu who inspired my pursuit of Electrical and Computer Engineering field as well as for his endless help and support throughout the masters program.

I also dedicate this work and give special thanks to Prof. Poovendran who is my source of inspiration and wisdom and who has encouraged me to pursue my dreams and finish my dissertation.

Lastly, I dedicate this dissertation to my three best friends Eric Darsenio Hunter, Paul Brennan and Ash Mahein who have supported me throughout the process. I will always appreciate all they have done, especially Eric Darsenio Hunter for his boundless help and support to find my foot in Seattle and helping me to develop work ethics. Moreover, I'm nothing but thankful to Paul Brennan and Ash Mahein for being there for me throughout the entire masters program. All of you have been my best cheerleaders.

Chapter 1

INTRODUCTION

The threats and security costs imposed to public and private sectors by cyber adversaries increase at an alarming rate with the ever growing interconnections between systems (e.g., Internet, Internet of Things (IoT) devices) [6, 12, 13, 18, 23]. The landscape of cyber threats vary from quick damaging attacks (e.g., most of the malware, computer viruses and Denial of Service (DoS) attacks) to long term, strategic, and stealthy attacks (e.g., Advanced Persistent Threats (APTs), Distributed Denial of Service (DDoS) attacks). End goals of cyber threats include data breaches, data hijacking and damaging systems infrastructures. The evidence from recent cyber threat intelligence reports show that most of the modern day cyber threats are often funded by resourceful state level actors and operate stealthily in the victim system for a long time in order to accomplish their end goals.

Mimicking normal system behavior (e.g., network traffic, process related event/system call traces) while operating in the victim system and low rate data exfiltration are few of the strategies employed by the current intelligent cyber attacks (e.g., polymorphic blending attacks [9], mimicry attacks [30]) to maintain stealthiness. Moreover, cyber attacks strategically change attack code base (binaries) to avoid getting detected by the cyber defenses that search for static signatures to identify cyber threats (e.g., Duqu 1.0 and Duqu 2.0). As a consequence, traditional cyber defenses such as moving target defenses, network intrusion detection systems, anomaly detectors and anti-virus software fail to detect modern day intelligent and stealthy cyber threats.

However, malicious activities initiated by any cyber adversary in a computer system introduce a set of related information flows to the victim system. Information flows consist

of data (e.g., file read/write, process[1] fork) or control commands (e.g., if/else statement) that exchange between different system components (e.g., processes, files, network sockets) [3, 15, 29]. Dynamic Information Flow tracking (DIFT) is a promising defense mechanism against cyber threats initially introduced in [29]. DIFT consist of three main components. 1) Tagging units which taint suspicious information flows that originate from untrusted sources (e.g., network sockets such as port80 for HTTP, processes of a third-party program). 2) Information flow tracker that keeps tack of the propagation of the tagged information flows in the systems. 3) Security check rules which are used to validate the authenticity of tagged flows when they are being used at a set of processes in the system (also known as traps or taint sinks).

DIFT has been employed as both offline and online cyber defense. Offline approaches widely used in cyber forensics first record all the information flows that arise during the run time of the host system as system logs and then replay the system logs in a different system/CPU to perform DIFT [5, 15, 20]. Online approaches rely on modified system hardware and/or software architecture to perform real time DIFT on a host system [17, 25, 29]. However, both offline and online DIFT approaches suffer from heavy memory and performance overhead added to the host system due to the tagging and tracking information flow capabilities required by the DIFT. It has been observed that DIFT can incur 2-20 times slowdown in the host system [7, 17].

One way to minimize the memory and performance overhead introduced by DIFT is to employ problem specific design of DIFT which only tag, track and security check on a smaller subset of information flows related to a prespecified class of cyber attacks [25, 31]. The authors of [25] designed DIFT that performs run time binary rewriting for automatic detection of overwrite attacks. The authors of [31] proposed an OS-aware DIFT to detect and analyze privacy-breaching malware. The authors of [27] proposed a low overhead information flow tracking method called LIFT that minimizes run-time overhead by exploiting

[1] A process is an instance of a computer program

aggressive dynamic binary instrumentation and optimizations. The authors of [16] proposed a DIFT architecture which is implemented using a coprocessor to reduce the memory and performance overhead.

LIBDFT [14, 17] is a widely used implementation of DIFT in systems security research that uses Intel Pin's Dynamic Binary Instrumentation (DBI) framework. LIBDFT provides a shared library of reduced overhead, reusable (can be used to detect wide range of cyber threats) DIFT that is applicable to many 32-bit single process commodity software. However, current version of LIBDFT is not compatible with the Intel Pin used by the most recent Linux distributions. It does not scale to perform DIFT on multiple programs. Moreover, it does not provide an ability to remotely start/terminate DIFT.

Therefore, improving the compatibility, scalability and usability of the LIBDFT shared library and proposing low overhead DIFT techniques will enable wide-range deployment of DIFT-based defense to protect modern computer systems from cyber threats. The contributions of this dissertation are the following.

1. We update the code base of LIBDFT library to support the latest version of Intel Pin, enabling multi program DIFT functionality to LIBDFT, and adds a RESTful service layer for improving the usability of LIBDFT (Chapter 2: RESTful DIFT).

2. We Propose a low memory and run time overhead DIFT technique that supports DIFT on multi process programs and provides a user friendly graphical visualization of the DIFT results (Chapter 3: Light weight DIFT).

3. We Validate the proposed DIFT techniques, RESTful DIFT and Light weight DIFT, using real world attack examples Screengrab attack and Multi-host Screengrab attack, respectively.

Chapter 2
PRELIMINARIES

This chapter provides the necessary backgrounds on the preliminaries required to follow this thesis. The remainder of this chapter is organized as follows. Section 2.1 discusses main components and different architectures of Dynamic Information Flow Tracking (DIFT). Section 2.2 introduces Intel Pin, a dynamic binary instrumentation framework that supports widely used DIFT implementations such as LIBDFT. Details of the LIBDFT is given in Section 2.3. Finally, limitations of the current LIBDFT implementation is discussed in Section 2.4.

2.1 Dynamic Information Flow Tracking (DIFT)

Dynamic Information Flow Tracking (DIFT) is used for tagging, tracking and security tag checking of spurious information flows entering a computer system through suspicious input channels. This is also known as the technique that co-relate with tagging and tracking of interesting data as they propagate during the execution of a program. Process of Dynamic Information Flow Tracking (DIFT) is characterized by three aspects:

1. Data sources

   Data sources are typically a program or a memory location where data of interest enter the system, generally after a function or a system call.

2. Data tracking

   Data tracking is the process of tracking tagged data during the execution of a program as they are copied and altered by program instructions.

3. Data sinks

   Data sinks are also program or memory locations where presence of tagged data can be checked to inspect or enforce data flow.

DIFT can be utilized to hardening software against malware, detection mechanism against zero-day vulnerabilities, cross-site scripting and buffer overflow attacks as well as detection and prevention of information leaks.

A tag is a flag that denotes the sensitivity of a dataflow. Input from untrusted sources is tagged as being potentially malicious by DIFT [29]. The tag status of the information flows propagates through the system based on the pre-specified propagation rules which are either data-flow based or data and control-flow based. Tagged flows are inspected by DIFT at locations called tag sinks, also referred to as traps, in order to determine whether a tagged information flow is malicious.

Logic related to Dynamic Information Flow Tracking can be injected in to the original program code in two ways:

1. Static Information Flow Tracking

   Static Information Flow Tracking requires DIFT logic to be added to the original source code during the development time and requires modifications to the compiler (See Figure 2.1).

2. Dynamic Information Flow Tracking

   Dynamic Information Flow Tracking is achieved during the run-time of a program. Meaning, Dynamic Information Flow Tracking logic will be injected into the original program instruction with the help of a Dynamic Binary Instrumentation (DBI) framework (e.g., Intel Pin, DynamoRIO) during the run-time of a program.

Both static and Dynamic Information Flow Tracking methods comes with their own advantages and disadvantages (see Table 2.1). DIFT is used throughout the work presented

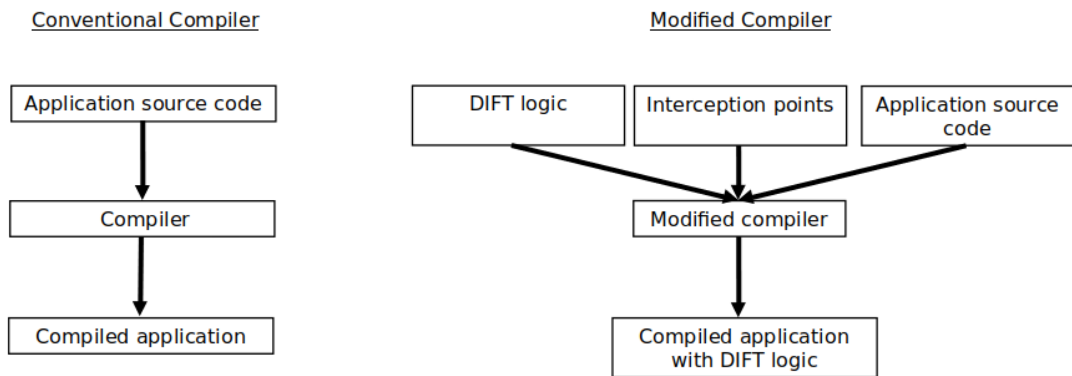


Figure 2.1: Conventional vs Static Information Flow Tracking (modified) compiler

in this dissertation.

2.2 Intel Pin

Intel Pin [4] is a dynamic binary instrumentation framework specifically catered towards IA-32 (32-bit), x86-64 (64-bit) and MIC (Many Integrated Core) instruction-set architectures that allow software developers to create their own dynamic program analysis tools. The application tools using Intel Pin is generally known as Pin tools which can be used to analyze user space applications running on Linux, Windows and Mac operating systems. Intel Pin tools performs instrumentation of a program binary file[1] at the run-time. Therefore, it does not require re-compiling of the original source code and can aid instrumenting programs that contain dynamically generated code.

Intel Pin provides a well structured and documented API (Application Programming Interface) that abstracts complex underlying instruction-set information and allows context details such as register contents to be passed into the pin tool's injected code as parameters. In addition, Intel Pin has the ability to automatically save and restore register values that are overwritten by the injected code such that application can continue to execute as originally

[1]Program binary is the result obtained once a program code is compiled and linked.

Dynamic Information Flow Tracking	Static Information Flow Tracking
Instrumentation code is added during the run-time of the application	Instrumentation code is added during the compilation of the application
Can be directly applied to any software	Can not be applied to any software
Can define a general set of rules for data flow tracking or set of rules can be developed specifically to a system/application	Application specific (Set of rules only works for one application)
Performance: Slow. Since it requires a DBI tool to be attached to the running application to inject DIFT logic	Performance: Faster than DIFT. Since, no DBI tool is required and DIFT logic is already bundled with the application

Table 2.1: Static vs Dynamic Information Flow Tracking

intended.

A pin tool consists of instrumentation, analysis and callback routines. Instrumentation routines are typically called when code that has not yet been re-compiled is about to execute and allows the insertion of analysis routines. In other words, instrumentation routines refers to the task of inspecting the binary instructions within a program to determine what analysis routines should be inserted where. Analysis routines are called when the code associated with them is executing. Callback routines are called when certain conditions are met, or certain events within the code has occurred.

Intel Pin performs program instrumentation by taking control of the program right after it loads into the system memory. Then JIT (Just-In-Time) compiler, re-compiles small sections of the binary code just before it executes. New instructions to perform analysis are added to the recompiled code which typically come from the Pin tool. Additionally, a vast array of optimization techniques are housed into to Intel Pin to obtain lowest possible run-time and

memory overhead. As of the time this dissertation was written Intel Pin's average overhead is around 30 percent without running a pin tool. However, the overhead can drastically depend on the quality of the pin tool written by the software developer.

2.3 LIBDFT

LIBDFT [14, 17] is a meta-tool[2] that acts as a shared library which implements Dynamic Information Flow Tracking using Intel Pin's Dynamic Binary Instrumentation (DBI) framework. LIBDFT provides an API for software developers to build DIFT-enabled pin tools that operates on unmodified program binaries running on commodity operating systems and hardware. In addition, its versatility and re-usability facilitate research and rapid prototyping.

LIBDFT was designed to be used with the Intel Pin DBI framework to create custom pin tools. LIBDFT utilize Intel Pin's virtual machine (VM) and an injector component that attach the VM to an already running process or a new process that launch itself. LIBDFT library rely on Intel Pin's extensive API to inspect and to modify instructions in a binary executable image. When a LIBDFT enabled Pin tool attaches to an already running process or launches a new process, the injector component of the Intel Pin first injects Intel Pin's run-time and then transfer the control to the LIBDFT enabled Pin tool.

Implementation of LIBDFT library consists of 3 major components (See Figure 2.2):

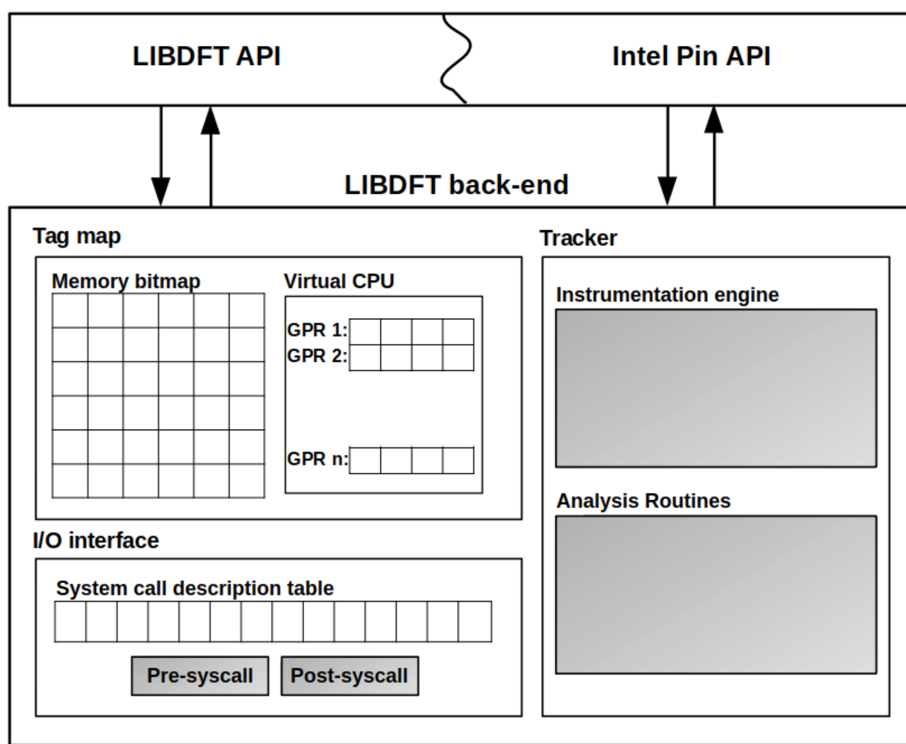
1. Tag map

2. Tracker

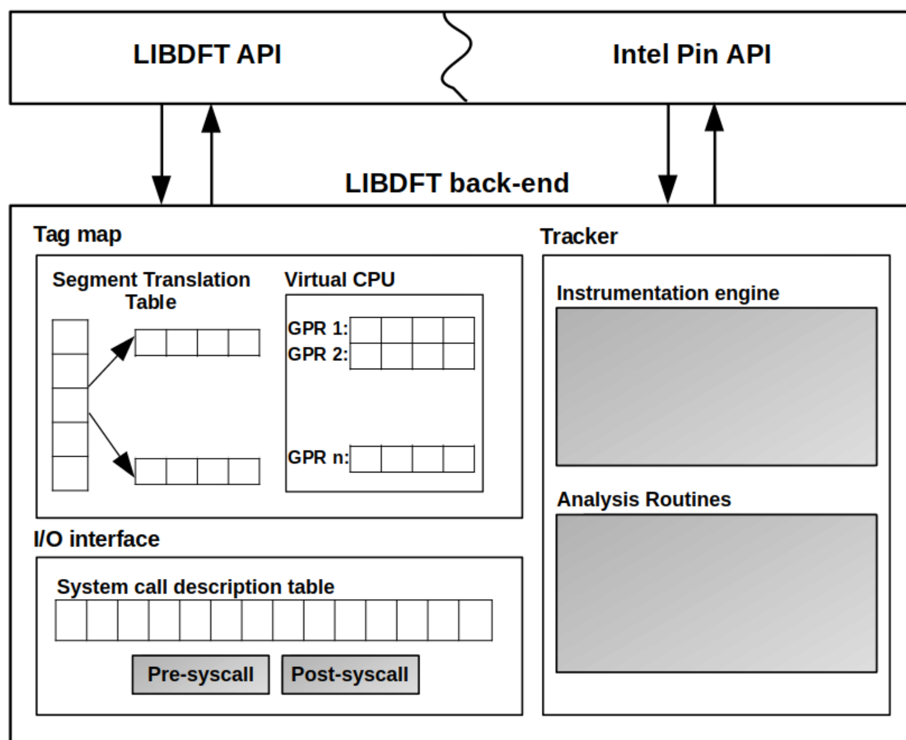
3. I/O Interface

Tag map is used to store data tags that consists of a process-wide data structure known

[2] Meta-tool is a tool that is used to make or modify another tool.



(a) LIBDFT architecture when configured with bit-sized tags



(b) LIBDFT architecture when configured with byte-sized tags

Figure 2.2: LIBDFT architecture [17]

as shadow memory[3] to maintain the tags of data stored in computer's main memory and CPU registers. The structure of the tags stored in the tag map is mainly governed by the ***granularity of the tagging*** and the ***size of the tags***.

When it comes to tagging granularity, LIBDFT can be configured to tag data as small as a single bit or larger contiguous memory chunks. Tagging using a single bit results in fine grained and accurate Dynamic Information Flow Tracking while using larger contiguous memory chunks results in more error prone Dynamic Information Flow Tracking. Nevertheless, picking exceedingly fine grained tagging comes with a significant memory usage since more memory space is needed to store the tags. For instance, bit-level tagging requires 8 tags for a single byte and 32 tags for a 32-bit CPU register. Therefore, LIBDFT uses byte-level tagging to overcome this memory hit. Using byte-level tagging makes more sense in modern CPU architectures since a byte is the smallest amount of addressable memory in modern CPUs. Furthermore, this method allows sufficiently fine-grained tagging for almost all the practical usages and it creates a good balance between performance and usability.

In terms of tag sizes, LIBDFT offers bit-sized tagging and byte-sized tags. The first allows creating memory conserving LIBDFT enabled Pin tools while to the second allows more sophisticated LIBDFT enabled Pin tools since software developers could use 8 different values to each tagged byte to represent different tag classes.

Furthermore, implementation of the tag map can be further broken down into following in-memory data structures:

1. Virtual CPU (VCPU)

   VCPU structure is utilized in LIBDFT to store tags for all 8 General Purpose Registers (GPRs) of a 32-bit CPU. The tag map holds various VCPU structures. Namely one VCPU structure for each thread that is created during a program execution. LIBDFT is implemented in such a way that it can capture thread creation and thread termina-

[3]Shadow memory is a technique that is used to track and store information on computer memory used by a program during its run-time. Typically shadow memory is invisible to the original/running program.

tion events and dynamically manage the number of VCPU structures during a given program run-time. In order to uniquely identify VCPU strictures associated with a specific thread a virtual ID is given by the LIBDFT. If LIBDFT is configured to use bit-sized tags, one byte of memory is used to hold four one bit tags that is necessary for each 32-bit GPRs. Therefore, memory required for each thread is 8 bytes. On the other hand, if LIBDFT is configured to use byte-sized tags, LIBDFT requires 4 bytes per each 32-bit GPR. Hence, 32 bytes are required per thread.

2. Memory bitmap (mem-bitmap)

   Mem-bitmap is utilized in LIBDFT to tag data residing within computer's main memory. This data structure is specifically used when LIBDFT is configured to use bit-sized tags. Mem-bitmap is a flat fixed sized memory that holds one bit for each byte of CPU addressable memory.

3. Segment Translation Table (STAB)

   Similar to the mem-bitmap structure STAB structure is utilized in LIBDFT to tag data resides in the main memory. Different from the mem-bitmap, STAB is used when LIBDFT is configured to use byte-sized tags. STAB stores tags in dynamically allocated tag map segments. At each time a program requests a chunk of memory through a system call such as mmap, malloc or shmat. LIBDFT intercepts the system call and allocates identically sized contiguous chunk of memory. During the process of initialization, LIBDFT allocates the STAB to map main memory addresses to their corresponding bytes since memory is given to a process in blocks known as pages. Therefore, each STAB entry points to a page. Importantly, implementation of the LIBDFT ensures segments that match with adjacent memory pages are adjacent as well. This technique helps to mitigate memory accesses crossing boundaries. When it comes to drawbacks, using byte-sized tags with STAB incurs high memory overhead.

Tracker is the core component of the LIBDFT library which is in-charge of instrumenting

a program to insert Dynamic Information Flow Tracking logic. Tracker consists of two components:

1. Instrumentation Engine

   Instrumentation engine is responsible for inspecting program's instruction to decide the analysis routines that should be injected. LIBDFT relies on Intel Pin's instrumentation engine to inspect each program instruction. LIBDFT first checks the instruction type (e.g., move, arithmetic or logic instruction), then it analyzes instruction's operands (e.g., memory address, register or immediate) to decide their category and finally instruction's length (e.g., word or double word). Once above mentioned information is gathered, LIBDFT relies on Intel Pin to insert appropriate analysis routine before each program instruction and instrumentation code is invoked once per every sequence of instruction.

2. Analysis Routines

   The analysis routines consist of the actual code that implements the Dynamic Information Flow Tracking (DIFT) logic for each instruction instrumented. Analysis routines are injected by the instrumentation engine before every instruction of the program. In comparison to the instrumentation code analysis routines execute more frequently. In other words, analysis code injected for a specific type of instruction will execute every time that type of instructions executes.

   Types of instructions instrumented by the analysis routines can be classified in to following classes depicted in Table 2.2.

I/O interface is responsible of handling the data exchange between kernel and the processes through system calls. I/O interface consists of a system call description table (syscall-desc) that holds information for all the 344 Linux system calls. It stores user defined call-backs and descriptors for the arguments for each system call as well as whether system

Instruction class	Description	Example instruction(s)
ALU	to calculate an arithmetic result	ADD, SUB, DIV, IMUL
XFER	Data transfers from a register to register, from register to a memory location and vice versa, from one memory location to another	MOV
CLR	to clear/zeroing registers	AND, XOR

Table 2.2: Classes of instructions analyzed by analysis routines

call writes data to the memory or reads data from the memory. When a system call is made during a program execution user defined call-backs are invoked upon entry (pre-syscall call-backs) and exit (post-syscall call-backs).

2.4 Limitations of LIBDFT

Although state of the art LIBDFT provides a promising mechanism for detecting cyber threats in computer systems, the wide range deployment of LIBDFT in consumer space is limited by the following factors:

1. Built on an older version of Intel Pin (version 2.7)

   Current version of LIBDFT only supports Intel Pin 2.7. However the most recent 32-bit Linux distributions and kernel versions are not supported by this version of Intel Pin. Therefore, it is required to update the code base of LIBDFT library to support latest version of Intel Pin and make use of LIBDFT to detect cyber attacks that targets modern Linux systems.

2. Only performs DIFT on a single program at a user specified time.

Current implementation of LIBDFT only tracks information flows that originates from a single user specified program instance during a single trial of LIBDFT execution. However, advanced cyber threats which targets computer systems such as Advanced Persistent Threats (e.g., Stuxnet, Duqu, Fancy Bear) and ransomware (e.g., WannaCry, Petya, CryptoLocker) often leverage multiple computer programs and/or multiple program instances (e.g., multiple running Linux terminals) in undertaking their attack steps. It has been observed that analyzing the behavior of a single program instance is inadequate to detect the presence of such advanced cyber adversaries due to the stealthy nature of those attacks [19, 15]. Therefore, using LIBDFT to perform security checks on the information flows arise from a single program instance can be often obsolete in successfully detecting the presence of advanced adversaries.

3. Does not have the ability to remotely monitor system logs or to start/terminate analysis on system wide processes.

   Current implementation of LIBDFT requires a skilled system administrator or security analyst to initiate and terminate the DIFT capabilities of LIBDFT as it requires knowledge of Linux terminal commands. Hence introducing a Graphical User Interface (GUI) on top of current LIBDFT code base can allow any user to easily initiate and terminate the execution of LIBDFT remotely in any device at will.

   Additionally, LIBDFT saves the program logs that record the violations during the program execution locally on the host. However, it has been recently observed that cyber adversaries delete or modify such program logs to hide their presence and maintain long-term operations in the victim host [2]. Hence, it is required to introduce an efficient mechanism which will transfer recorded program logs to a secure centralized server to protect them from getting deleted or modified by the cyber adversaries.

4. Can only analyze single process programs

   Current version of LIBDFT can only perform DIFT on programs that execute through

a single process in the system. However, modern CPU intensive computer programs (e.g., Mozilla Firefox, Adobe Photoshop) utilize multiple processes to execute their workload efficiently. Using DIFT on such multi-process programs require memory efficient modifications to the current LIBDFT code base.

Chapter 3

RESTFUL DIFT

This chapter mainly focuses on improving the design and architecture of the LIBDFT library. RESTful DIFT was designed to address three main limitations (1-3 in chapter 2.4) that come with LIBDFT library and its architecture. Since LIBDFT was designed with older version of Intel Pin (version 2.7) and Linux kernels in mind, it faces compatibility issues with latest Linux distributions and Linux kernel versions. In order to address the compatibility issue, RESTful DIFT was implemented with the latest version of Intel Pin (version 3.18) and updated implementation of LIBDFT at its core to support the latest version of Intel Pin. Thus, RESTful DIFT is more robust, backward and forward compatible with 32-bit versions of newer and older Linux distributions as well as Linux kernels.

On the other hand, due to LIBDFT's design limitations, current state of the art LIBDFT can only perform Dynamic Information Flow Tracking (DIFT) on a single program at a given time. RESTful DIFT addresses this limitation by adding two new components to its core called **starter script** and **wrapper component** (Discussed in 3.3.1) which automates system process discovery and analyzes them when they spawn by creating instances of RESTful DIFT in the background for each spawning process. Thus, RESTful DIFT does not require any user interaction to manually hand over programs to run analysis on nor is it limited to analyzing one program at a given time.

Compared to LIBDFT, RESTful DIFT pushes Dynamic Flow Information Flow Tracking more into the consumer space by implementing easier ways to interact and to analyze process logs created during the system process analysis. For instance, current state of the art LIBDFT based Dynamic Information Flow Tracking tools requires users to have a good knowledge about Linux commands where as RESTful DIFT does not. This is achieved by

the REST service layer (Discussed in 3.3.2) that sits on top of the DIFT core of the RESTful DIFT, allowing users to interact with the DIFT core using a web dashboard.

The remainder of this chapter is organized as follows. Section 3.1 introduces REpresentation State Transfer (REST) architecture. Section 3.2 explains collecting process notifications through Linux kernel interface (Netlink API). Details of the RESTful DIFT architecture design is given in Section 3.3. Section 3.4 presents case study results where a sample attack (Screengrab) is analyzed using the RESTful DIFT. Finally, limitations of the RESFTful DIFT is discussed in Section 3.5.

3.1 REpresentational State Transfer (REST) architecture

REST is a software architectural style that provides communication standard and set of rules between two or more computer systems residing on the web (See Figure 3.1). Web services or computer systems that obey the REST software architecture style are known as RESTful web services. Interactions with REST based web services happens through 4 types of Hypertext Transfer Protocol (HTTP) requests (see Table 3.1). REST software architecture was first introduced by Roy Fielding[8] in 2000.

HTTP request type	Description
GET	Retrieve a specific resource or a collection of resources from server
POST	Create a new resource in the server
PUT	Update a specific resource in the server
DELETE	Remove a specific resource in the server

Table 3.1: Different types of HTTP requests

Similar to other software architectural styles, REST architecture has 6 guiding principles that must be satisfied if a web service needs to be referred as RESTful.

1. Uniform Interface

   This is a key constraint which differentiate RESTful service from other software architectural styles. Uniform interface defines there should be a uniform way of interacting with a given RESTful server regardless of device or type of application.

2. Stateless

   Services that obey REST architecture must be stateless. To elaborate, the server does not need to know anything about the state of the client device and vice-versa. In REST architecture, client must include all the information for the server to fulfill a request. Statelessness of RESTful architecture enables greater availability by allowing many clients to be served in a given time since the server does not have to store or maintain, update details regarding a session state[1].

3. Cacheable

   Every response within a RESTful enabled system should include whether the response to a request is cachable or not and the duration response can be cached at the client-side. If a response is cachable, then then the client has the right to reuse the response data to a request for later use.

4. Client-Server

   REST enabled applications must maintain client-server division. A client is a device that requests resources by sending HTTP request(s) to a server. A server is a device that holds resources and has the ability to serve a client with needed resources according to the request type. Client-server division within REST architecture improve the portability of the client logic across multiple platforms and improve scalability of the components within the server.

[1] Session state is a method that keep track of the a user information during a series of HTTP requests. Session state allows software developers to store data about users as they navigate through web application.

5. Layered System

   REST application must consist of multiple layers. However, each layer does not need to know any working details or implementation details of any other layer within the system. This rule hardens the security of REST compliant systems.

6. Code on Demand

   This is an optional rule. Code on demand stands for, server can provide executable code to the client. This allows client side implementation to be simplified by reducing the number of features to be pre-implemented.

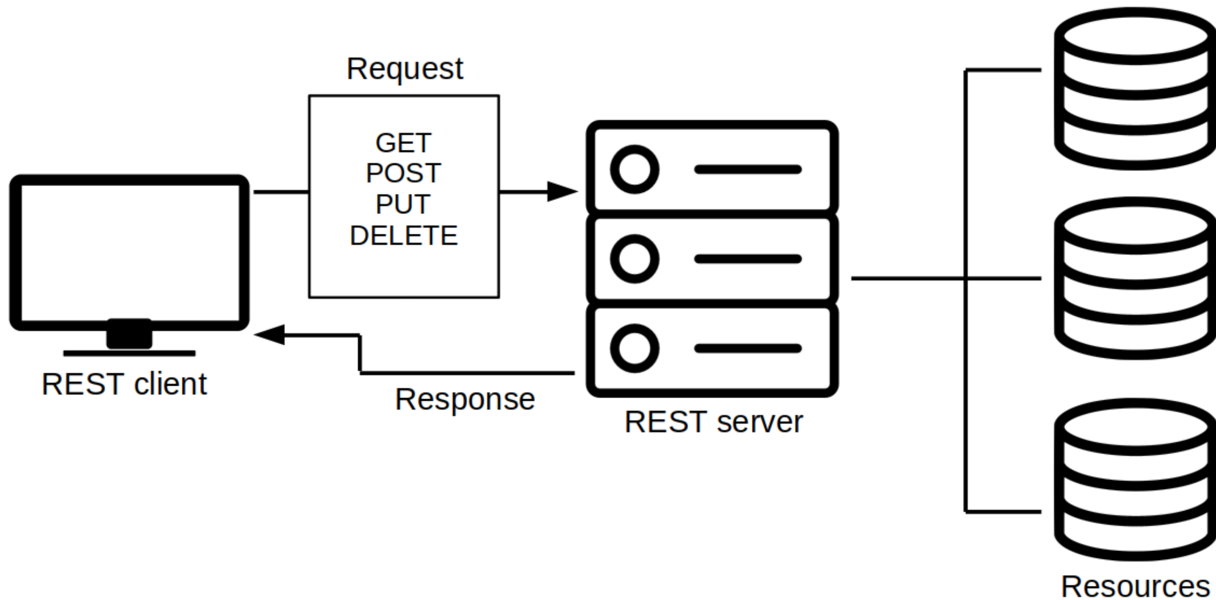


Figure 3.1: RESTful architecture

3.2 Process notifications through Netlink API

Netlink API is a Linux kernel interface that is used for inter-process communication (IPC) between both kernel space processes and user-space processes and between user-space pro-

cesses. Netlink API's IPC is achieved by using a standard socket based interface to transfer data between user-space processes and processes related to kernel modules.

For the purpose of RESTful DIFT implementation, Netlink API is used to connect to the Linux kernel's process event module to receive process creation notifications during program executions such that captured processes can be analyzed via LIBDFT. A process notification received through Netlink API consists of three C language structures[2]:

1. Netlink header
2. Kernel connector
3. Process event connector

Fields of the above structures are depicted in Figure 3.2 along with their field descriptions in Table 3.2.

3.3 RESTful DIFT architecture

RESTful DIFT was developed to bring Dynamic Information Flow Tracking based on Intel Pin and LIBDFT to the consumer space by allowing the end-users to remotely analyze and manage process logs of running processes in target systems in real-time. In addition, RESTful DIFT allows end-users to combine target systems into groups and manage them through a web-enabled dashboard.

Architecture of the RESTful DIFT mainly consists of two components (See Figure 3.3), namely:

1. DIFT core
2. RESTful service

In-depth details on each components are discussed in subsections 3.3.1, 3.3.2 respectively.

[2]Structure in C language is a user defined data type that allows to combine data items of different types.

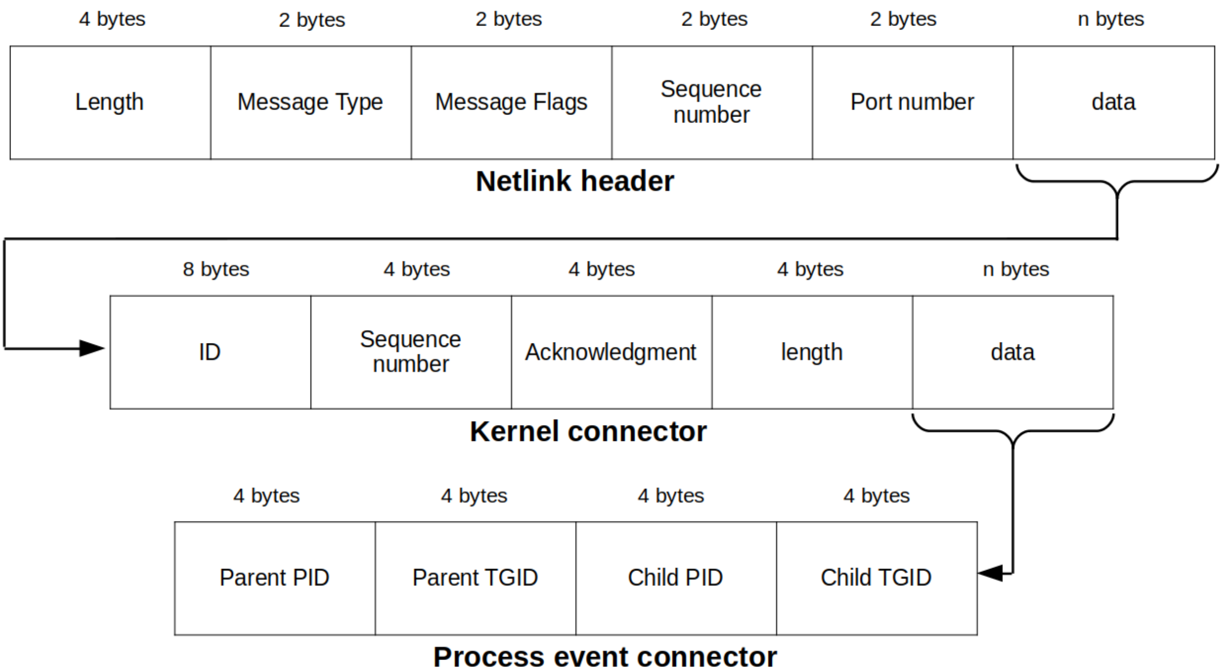


Figure 3.2: Netlink header, kernel connector and process event connector structures

3.3.1 DIFT core

DIFT core is the core architectural component that is responsible of capturing and analyzing processes within a target system once the target system boots up. DIFT core consists of three sub-components (See Figure 3.3):

1. Start script

   Start script is a Linux bash script that helps to launch wrapper component at Linux operating system boot. Start script resides within "/etc/init.d" directory within the Linux file system. Any bash script stored in this location will be started during the boot process of a Linux operating system. This feature was utilized in RESTful DIFT to launch the wrapper program via start script during the Linux operating system boot-up.

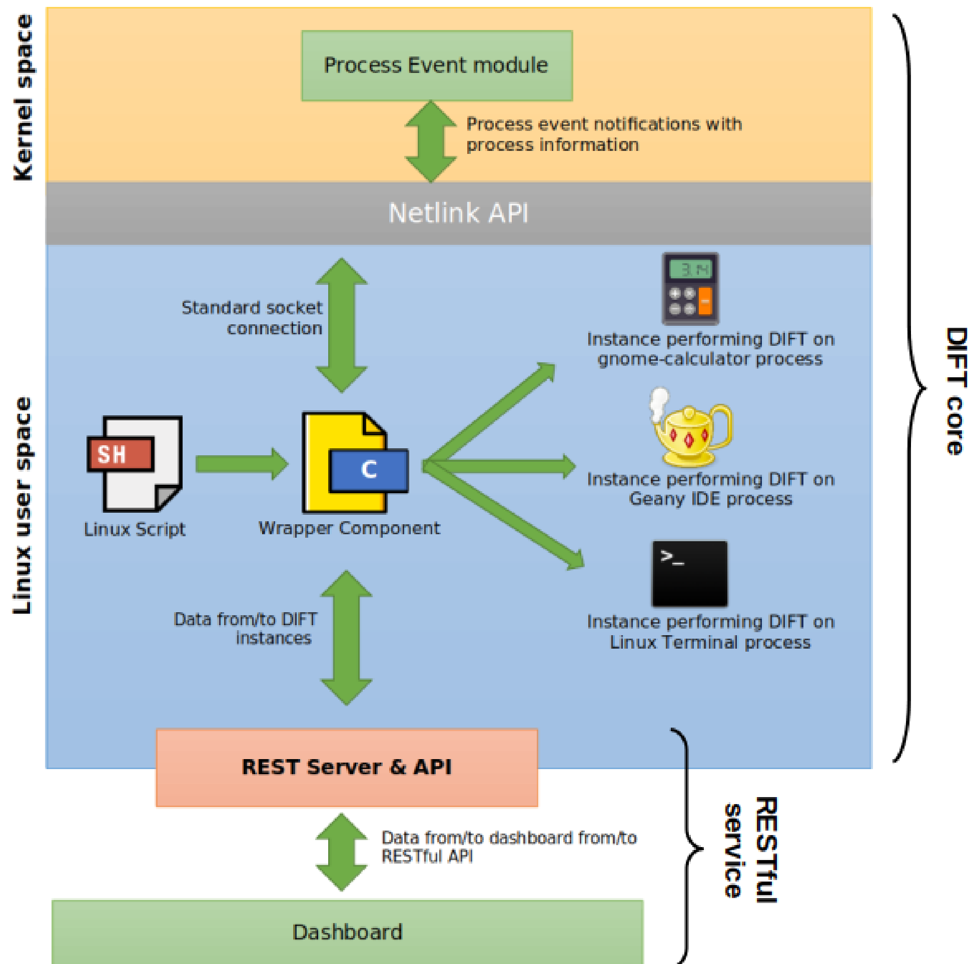


Figure 3.3: Architecture of the RESTful DIFT

2. Wrapper

   Wrapper component is a C program that connects with Netlink API of the Linux kernel via standard socket-based interface to receive notifications on new process creation. Upon detection of new processes wrapper component will launch instances of Analysis tool attached to the discovered processes.

   In addition, wrapper component is responsible for creation of shared memory buffers between analysis tool instances to retrieve process logs and managing them by giving

each instance created an unique numerical identifier. Wrapper component communicates with DIFT instances via Inter-processes communication protocol using a 100 bytes shared memory buffer resides between each DIFT instance and the wrapper component. Inter-process communication protocol between the wrapper program and the DIFT instances is implemented as a one-way communication protocol, meaning DIFT instance could only write to the shared memory buffer while wrapper program could only read from the shared memory buffer. This decision was made to harden the security of the over-all communication protocol such that attackers can not write into the shared memory buffer through REST communication protocol to alter the state of the DIFT instances. Furthermore, Linux kernel APIs were used for the implementation of the Inter-process communication channel and the 100 bytes shared memory buffer. This decision was made to ensure forward and backward compatibility with newer and older Linux kernels and distributions.

In addition to the above mentioned tasks, wrapper also implements client side REST logic to send retrieved process logs from the DIFT instances' shared memory buffer to the REST server. REST client logic was implemented using the Linux's CURL[3] library. This decision was also made to ensure forward and backward compatibility of RESTful DIFT with newer and older versions of Linux kernels and distributions.

3. Analysis tool instances

   Analysis tool is built using the latest version of Intel Pin (version 3.18) and updated LIBDFT library to support Intel Pin version 3.18. Analysis tools are responsible for tagging, tracking and security tag checking of spurious information flows entering through suspicious input channels. Input from untrusted sources are tagged and the tag status propagates during the program execution based on the pre-specified tag propagation rules which are either data-flow-based and control-based. Tagged flows

[3] CURL is a software project that provides library (libcurl) and command line tool utilities to transfer data using different network protocols such as HTTP, FTP and SMTP

are inspected by the analysis tool at locations called tag sinks, also referred as traps, in order to determine whether a tagged information flow is malicious.

Once an malicious information flow is detected, analysis tool will hold the execution of the program. Then notify the user through command-line interface of the target system along with the memory address of the instruction that is executing on the malicious data flow and the memory address of the instruction that executing instruction will branch to. Once the user is notified, user could grant access to the program for further execution or deny access for further execution.

All the violations captured during the analysis is stored locally as a log file as well as transferred to the wrapper via inter-process communication protocol such that log information can be transferred to the REST server to be viewed on the user dashboard (discussed in 3.3.2).

3.3.2 RESTful service

RESTful service implements the communication protocol that enables end-users to analyze and manage process logs of running processes in a target system as well as managing devices by combining them into groups. RESTful DIFT's REST service component can be broken down into two sub components:

1. REST server

   REST server provides an interface between wrapper component and the user dashboard through a web API. Web API was developed using Python3 and Flask. Flask is a lightweight web application framework, with the ability to scale up to a complex web applications. In a technical stand point, Flask implements the REST enabled API routes (See Table 3.4) that defines an interface between the wrapper and the user dashboard. Flask was chosen to implement the REST server due to its lightweight code base, simplicity and extendibility which allows scaling and adding new features

to the REST server effortless.

In addition, REST server implements a database service using MongoDB. MongoDB is a database management system which is document-oriented[4] used for high volume data storage applications that fits the requirements of RESTful DIFT due to vast amount of process logs stored for each process executing in a target system. Mongo database used in RESTful DIFT consists of 3 collections (Similar to a table in SQL database):

- Groups
- Devices
- Analysis processes

that allows RESTful DIFT to manage process logs of running processes as well as to combine devices into group for the ease of managing them. Detailed descriptions of each collection and depiction of relationships with each collection is shown in Table 3.3 and Figure 3.4 respectively.

2. User dashboard

User dashboard (see Figure 3.5) implements the client side of the RESTful service. User dashboard was built as a web application using HTML, CSS and JavaScript. Therefore, user dashboard is platform independent where users can monitor multiple systems through one dashboard using device of their choice. User dashboard allows end-users to visualize all the processes being analyzed by the DIFT core on a target system. In addition, end-users can start or terminate analysis of a process(es) through the dashboard, read process logs. Finally, if the process being analyzed by the DIFT core utilizes ports for communicating with internet or for inter-process communication to communicate with another process end-users can command DIFT core to analyze

[4]Document-oriented databases are kind of databases that store information in JSON like data structures rather than using tables.

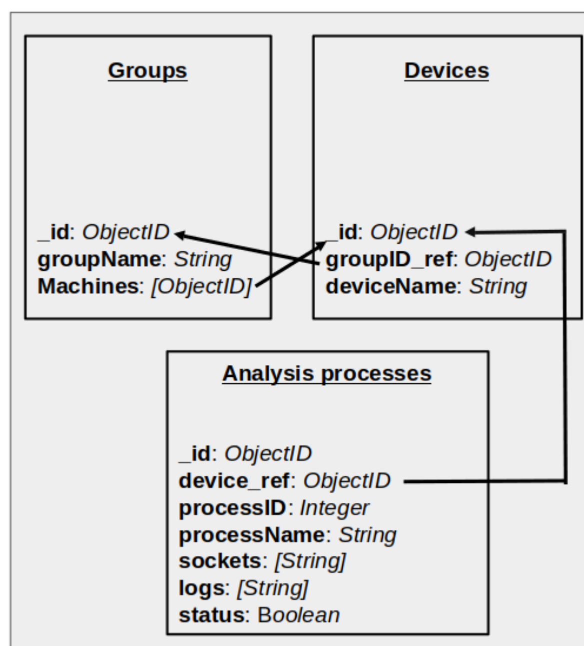


Figure 3.4: Database collections and relationships between them.

all the ports the process is utilizing (default option) or select specific ports through a list of ports used by the process.

3.4 Analysis of Screengrab attack using RESTful DIFT

Screengrab is a malicious program written in C language which has the ability to take screenshot(s) of the host computer without the user's permission. Screengrab has the ability to occasionally take screenshot(s) of the victim system and send selected screenshot(s) to an attacker specified server. For the purpose of analysis scenarios presented thorough out the dissertation work, screengrab was configured to send the captured screenshot(s) to the local-host's (http://127.0.0.1) "/home/kalana/Desktop" directory along with the file name "attack" appended with the time stamp when the screenshot was taken.

Example: `http://127.0.0.1/home/kalana/Desktop/attack_2021-04-20_11:28:06`

Analysis of screengab attack consists of 3 scenarios:

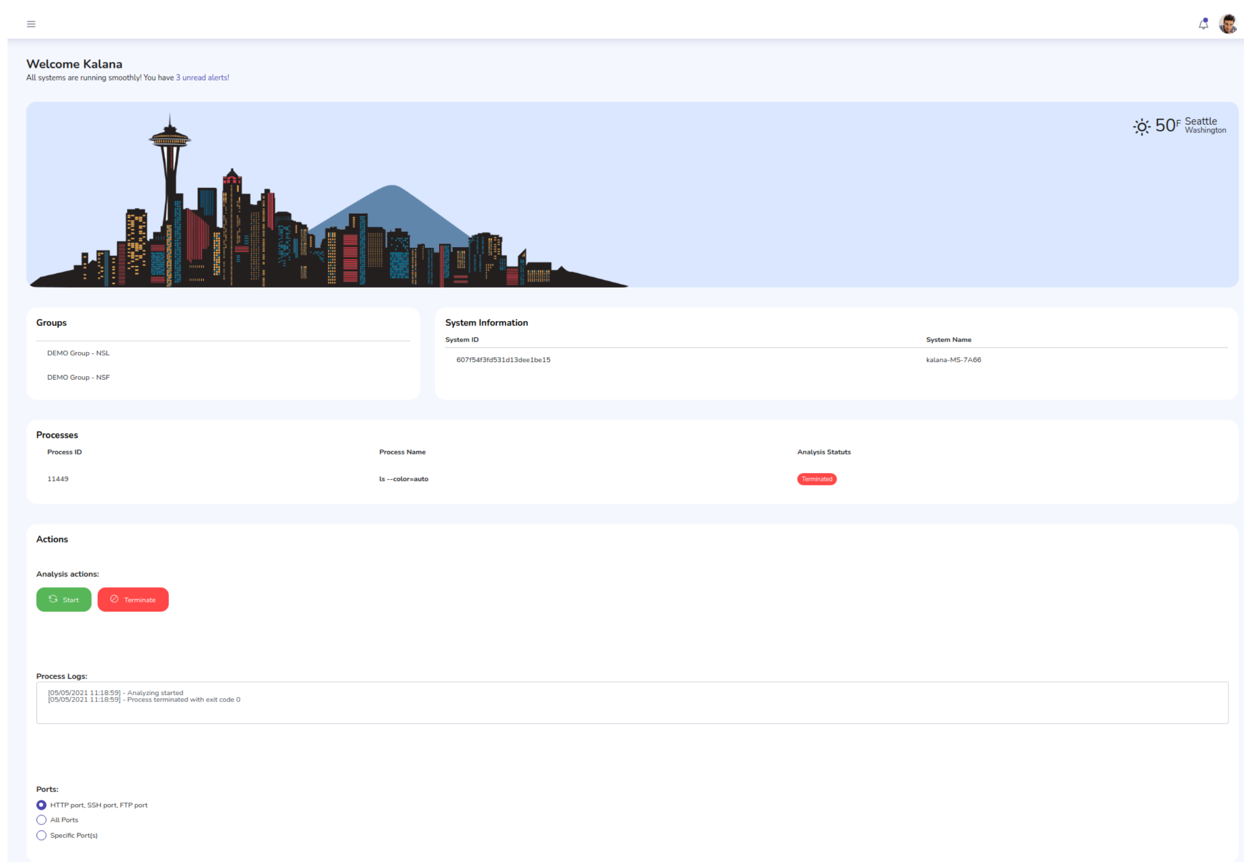


Figure 3.5: User dasboard of the RESTful DIFT's RESTful service component

1. Scenario 1: Attack without RESTful DIFT analysis

   This scenario is meant to show the behavior of screengrab attack. During this scenario screengrab attack was executed with RESTful DIFT disabled. The screengrab attack was executed using following command on the Linux terminal

   ./screengrab /home/kalana/Desktop/attack

   The first command line argument (./screengrab) represents the screengrab executable while the second argument (home/kalana/Desktop/attack) represents the path and the file name of the screenshot is saved within the host computer.

   Upon completion of the attack, an image file (captured screenshot) is created under

the above mentioned directory of the localhost (in this case local-host is the victim computer) with a timestamp.

2. Scenario 2: Attack with RESTful DIFT – User denies execution upon offending instruction detection

   During this scenario screengrab attack was executed while RESTful DIFT was running in the victim computer. Once the attack execution was detected, RESTful DIFT binds to the attack process and start analyzing data flows and instructions that data flows were used. During the execution of the attack process, the execution of the attack was halted by the RESTful DIFT and the user was prompted with the offending instruction memory address as well as memory address of it's branch target. Once the attack process is halted, the end-user has the ability to allow further execution of the screengrab process by typing "Yes" or deny further execution by typing "No" on the command prompt (See Figure 3.6). For the purpose of this scenario, an assumption was made that the end-user will deny further execution of the screengrab process, resulting RESTful DIFT to deny the further execution of the screengrab attack process. Thus no image file (screenshot) was created in the local-host's "/home/kalana/Desktop" directory.

   During the execution of the screengrab attack code (See Appendix A), variable argv (array of terminal arguments) was tagged by the RESTful DIFT since its a stream of data which is entering the system through command prompt. Then this tag is propagated to the "filename" character pointer variable due to argv[1] pointer being copied to it. Moreover, during the execution of line 46 of the code, character array `"file_name_with_time_stamp"` was tagged since "filename" variable was used to construct the string with the time stamp stored in the `"file_name_with_time_stamp"` variable. Then, during the execution of the fopen function at line 50, variable "fp" (file pointer) was tagged since
   `"file_name_with_time_stamp"` was passed in to the function as a parameter. Also,

variable `"png_ptr"` was tagged due to function `"png_init_io"` directly using value stored in “fp” to initialize the `"png_ptr"`. Finally at line 92 which is a system call to write the image header information to the disk, RESTful DIFT halts the program from executing due to tagged data flow being used in a system call, resulting in the prompt to grant or deny further execution of the program from end-user.

3. Scenario 3: Attack with RESTful DIFT – User allows execution upon offending instruction detection

   During scenario 3, screengrab attack was executed while RESTful DIFT was running in the victim computer. Similarly to the scenario 2, execution of the screengrab attack was halted and requests the end-user for permission for further execution at the line 92 (See Appendix A) due to tagged data flow directly used in a system call. Different from scenario 2, end-user will grant access to the screengrab process for further execution by typing “Yes” (See Figure 3.6). Thus, screengrab will continue execution and save an image file (screenshot) in local-host’s “home/kalana/Desktop”.

3.5 Limitations

RESTful DIFT improves the scalability, usability and reliability of LIBDFT through enabling DIFT on multiple programs at a specified time, providing support for modern Linux systems and allowing users to remotely start and terminate DIFT on system processes. However RESTful DIFT suffers from the following limitations.

1. Currently limited to 32-bit Linux operating systems

   Since the code base of RESTful DIFT is built on the LIBDFT code base, the current version of RESTful DIFT can only perform DIFT on 32-bit Linux operating systems. Even though, it only requires 384 megabytes in total to analyze 32-bit programs, it would require 32 terabytes to analyse 64-bit based programs using RESTful DIFT

```
kalana@kalana-desktop: ~/Desktop/libdft_32/tools
File Edit View Terminal Help
kalana@kalana-desktop:~/Desktop/libdft_32/tools$ sudo $PIN_HOME/pin -follow_exec
v -t libdft-dta.so -l /home/kalana/Desktop/log.txt -- /home/kalana/Downloads/scr
eengrab/screengrab /home/kalana/Desktop/attack
Size of tagmap --> 1
|---------------------------------------------------------------|
| Suspicious instruction found at memory address   : 0xb776428b |
| Supsicious instruction will branch instruction at: 0x95985a00 |
|---------------------------------------------------------------|

Allow type [Yes] deny type [No]:
```

Figure 3.6: RESTful DIFT prompt to grant user's permission for further execution of the program

as LIBDFT's code base is only optimized for 32-bit programs. This makes analysis using LIBDFT unrealistic in 64-bit operating system space. Therefore, deployment of RESTful DIFT in 64-bit Linux systems requires modifications to the LIBDFT code base shared by RESTful DIFT.

2. Does not provide any information on set of events that leads to a security violation

   Identifying causal dependencies of a set of events that occurred during the system execution is infeasible in the instruction level (e.g., arithmetic operations, register/memory move operations) as instruction level activities do not provide human explainable information on event correlation. However, information on a set of events that lead to

a violation can provide better decision making capabilities for the user on terminating the underline process or let it run. For example, consider a scenario where a security violation occurs at a process A due to a command executed by a trusted/secured process B in process A. In such case RESTful DIFT user will not know the violation at process A is initiated by process B and might have to terminate the process A.

3. Can only analyze single process programs

   Current version of RESTful DIFT can only perform DIFT on programs that executes through a single process in the system. This limitation is inherited from LIBDFT as RESTful DIFT is build upon the LIBDFT code base. However modern CPU intensive computer programs (e.g., Mozilla Firefox, Adobe Photoshop) utilize multiple processes to execute their workload efficiently. Using DIFT on such multi-process programs requires memory efficient modifications to the core LIBDFT code base shared by RESTful DIFT.

Field name	Description
Netlink header	
Length	The total length of the message in bytes including the data section
Message Type	Specifies the type of data the message is carrying
Message Flag	Used to modify the behavior of the message type
Sequence Number	Used to refer to a previous message. This field is optional
Port Number	Specifies the peer to which the message should be delivered
Data	Holds the kernel connector structure. This is a variable length field
Kernel connector	
ID	A unique identifier that identifies a kernel connector structure
Sequence Number	Similar to Netlink's sequence number field used to refer to a previous kernel connector structure
Acknowledgment	Defines if a reply is expected from the receiver.
Length	The total length of the structure including the data section
Data	Holds the process event connector structure. This is a variable length field
Process event connector	
Parent PID	Parent process ID of the current process
Parent TGID	Parent process thread group ID of current process
Child PID	Current process's process ID
Child TGID	Current process's thread group ID

Table 3.2: Field descriptions for Netlink header, kernel connector and process event connector structures

Collection name	Field name	Type	Description
Groups	_id	ObjectID	A unique identifier to uniquely identify a group that registered in the database
	groupName	String	Stores the name of the group
	Machines	Array of ObjectIDs	Stores a collection of device object IDs that are owned by a group
Device	_id	ObjectID	Used to uniquely identify a device registered under a group
	groupID_ref	ObjectID	Stores a copy of the group object id the device belongs to. This backreference helps for fast database queries when trying to retrieve information about the group a device belongs to.
	deviceName	String	Stores name of the device
Analysis processes	_id	ObjectID	Used to uniquely identify a process analyzed within a device
	device_ref	ObjectID	Back reference to the device a process belongs to
	ProcessID	Integer	Process ID of the process (Given by the operating system)
	processName	String	Name of the process
	sockets	Array of strings	Holds a list of ports that needs to be analyzed
	logs	Array of strings	Holds process logs
	status	Boolean	True - Analysis terminated False - Analysis still running

Table 3.3: Database collection fields and their descriptions

Route	Type	Description
/groups	POST	Adds a new group to the system
/groups	GET	Retrieves all the groups in the system
/devices/id	POST	Adds a device to a group using group ID
/devices/name	POST	Adds a device to a group using group name
/devices/<group_obj_id>	GET	Returns all the devices under a group, given group ID
/analysisProcess	GET	Adds an analysis process to the device given device ID
/analysisProcess/<device_obj_id>	GET	Retrieves all the analysis processes under a device given device ID
/analysisProcess/terminate/ <process_u_id>	GET	Terminates an analysis process given process ID. Intended to use from dashboard client
/analysisProcess/markterminated/ <device_id>/<process_id>/ <exit_code>	GET	Set the terminate status within the database and adds exit status to the process log. Intended to use from wrapper program client

Table 3.4: RESTful DIFT API routes and descriptions

Chapter 4

LIGHT WEIGHT DIFT

This chapter focuses on the Light Weight DIFT implementation. From an architectural stand point Light Weight DIFT was designed to be portable and light weight in computational requirements. Light Weight DIFT was implemented to improve the limitations of the LIBDFT since it does not rely on the LIBDFT library and its heavy code base for program analysis. Therefore, Light Weight DIFT does not inherit LIBDFT's implementation limitations (Chapter 3.5) such as inability to perform analysis on multi process programs, being limited to 32-bit Linux distributions and kernel versions and inability to provide any information on set of events that leads to a security violation.

Light Weight DIFT's reliance on Intel Pin is minimal (Only used in the Intel Pin based logging tool discussed in 4.3). This decision was taken to minimize the performance overhead as much as possible that comes along with the Intel Pin due to its instruction injection to the executing program's original code base. Instead, Light Weight DIFT mainly relies on Linux kernel components such as Netlink and process event module to establish communication with the kernel space and to receive process event notification (such as process creation within Linux operating systems) respectively, allowing Light Weight DIFT to maintain its light weight nature and portability.

Moreover, Light Weight DIFT's visualization tools (Discussed in 4.3 and 4.3.2) allows security analysts to draw a clear picture of the behavior of an malicious program through process trace graphs which not only depicts the behavior of a malicious program on the victim's computer but also on adversary's computer.

The remainder of this chapter is organized as follows. Section 4.1 introduces Neo4j. Section 4.2 and 4.3 discuss two major components of the Light Weight DIFT's architecture

along with in depth implementation details of the each component. Finally, Section 4.4 presents results of a case study where a sample attack (Multi-host screengrab) is analyzed using the Light Weight DIFT.

4.1 Neo4j

Neo4j [1] is an open-source and NOSQL graphing database that provides an ACID-compliant[1] back-end for applications. Neo4j is a native graphing database since it methodically implements the property graph model directly down to storage level. In other words, the data is stored exactly the way developers whiteboard it.

In a property graph model based database management system such as Neo4j data is stored as nodes, relationships and properties (i.e. data stored on nodes and relationships). Nodes are the data entries of the graph which can hold any number of attributes as key-value pairs known as properties. Relationships in a property model based database management system provides directed, named, semantically-directed connections between two nodes. A relationship always consists of a type, a direction, a start and an end node. In most cases relationships consists of quantitative properties such as cost, weight, distances, etc.

Due to Neo4j's flexible data model, high scalability, easy retrieval of data through cypher query language and its real-time visualization abilities Neo4j was chosen as the database management system for the Visual DIFT.

4.2 View from the victim's perspective Light Weight DIFT

View from the victim's perspective Light weight DIFT was implemented to capture the behavior of processes during an execution of a malicious remote attack taking place through

[1] ACID stands for 4 properties known as atomicity, consistency, isolation and durability which ensures that database transaction in timely manner. Any database management system that complies to mentioned 4 properties, they are ACID-compliant.

SSH[2] and SCP[3] communication protocols. View from the victim's perspective light weight DIFT helps to visualize process trace of a victim system during the execution of a SSH and SCP based attack. Implementation of the victim-side light weight DIFT consists of three components discussed in 4.2.1, 4.2.2, 4.2.3.

4.2.1 Start and termination script

This component will first start the logging program on the victim side just right before it starts the attack. This is achieved by sending a signal to the victim computer through SSH communication protocol to start the Netlink based logging program discussed in 4.2.2. Once the attack is finished start and termination script will send another signal through SSH communication protocol to the victim computer to stop the process logging in 20 seconds once the signal is received. This 20 second padding time is added to give processes related to the attack created on the victim side to finish their tasks in case they were still executing during the time the signal is received. This synchronization mechanism (start and stop signals) added to clearly capture processes created during the time frame of the attack.

4.2.2 Netlink based logging tool

Netlink based logging tool is written using C/C++ to directly connect with the Linux kernel using Netlink kernel module which will allow to get process notifications along with their PID (Process ID), PPID (Parent Process ID) and executable details. At the end of the execution of the Netlink based logger, it will produce a log file containing all the processes executed in the victim system during the time frame of the attack. Then captured information related to the processes spawned in the victim system will be fed into the visualization tool discussed in 4.2.3 for further analysis.

---

[2]SSH or Secure Shell is a remote administrative protocol that helps users to login to an another system over the internet.

[3]SCP or Secure Copy Protocol helps to transfer files from one computer to another over the internet.

4.2.3 Visualization tool

Visualization component is a Python based script that acts as a post processing tool which takes the log file produced by the logging tool discussed in 4.2.2. Python script will read through the log file and first find the log information where the stop signal was received. Then it will take all the process log information up-to to the time frame when the termination signal was received and creates in-memory dictionary objects for each unique process. Upon completion of in-memory dictionary objects, python script will create relationships between parent and children processes and send these information through Neo4j Python driver to the Neo4j Database management system to produce process trace graph.

4.3 View from the adversary's perspective Light Weight DIFT

View from the adversary's perspective Light Weight DIFT was implemented to give security analysts a clear picture of the processes created on the adversary's side of the system during the attack. Especially to find out how processes are created by a malicious program during its execution. This tool helps to visualize process trace of a malicious program during its execution. View from the adversay's perspective Light Weight DIFT consists of two components discussed in 4.3.1, 4.3.2.

4.3.1 Intel Pin based logging tool

Intel Pin based logging tool is a C/C++ program built using Intel Pin tool which allows to bind to a malicious program and detect all the processes spawned during its run time. During creation of a new processes, Intel Pin based logging tool captures following information of a spawning process:

1. Commands executed by the process along with the arguments passed into the command.

2. User which the process belongs to

3. Group which a user belongs to

Above information will be logged by the logging tool to a log file during the analysis of a malicious program which will be then fed to the visualization tool discussed in 4.3.2 for post processing.

4.3.2 Visualization tool

Visualization component is a python based script that acts as a post processing tool which takes the log file produced by the logging tool discussed in 4.3.1 and produces in-memory python dictionary objects for each unique process created during the execution of a malicious program along with their children processes. Ultimately, these in-memory dictionary objects are fed to the Neo4j graphing database management system through Neo4j python driver for visualization.

4.4 Analysis of Multi-host Screengrab attack using Light Weight DIFT

Multi-host screengrab attack is an extension of the screengrab attack. While screengrab attack is only limited to infecting a single target system, multi-host screengrab attack has the ability to infect all the computers in a computer network once it infects a single computer in a network. During the multi-host screengrab attack, we assume that adversary will first grant access to one of the computers within the network without user's knowledge and transfer all the files and scripts needed for the attack to infect all the computers within the network. Once initial step was completed, multi-host screengrab attack will discover all IP addresses of the computers within the network and send the screengrab attack payload to "/tmp" directory of all the computers in the network via SCP without users knowledge. Then from the computer that the adversary first grants access to, open SSH sessions to all the computers and the attack follows following steps during its execution:

1. Install dependencies on victim computers needed to execute the attack without users' knowledge

2. Unzip the screengrab attack payload within "/tmp" directory in the victim computers

3. Compile the screengrab attack code

4. Execute the screengrab attack

Upon completion of all the steps above, screenshot file will be created in the desktop directory of all the victim computers within the network.

Analysis of multi-host screengrab attack consists of two scenarios namely, victim-side analysis and adversary-side analysis discussed in 4.4.1 and 4.4.2 respectively.

4.4.1 Victim-side analysis

Victim-side analysis of the multi-host screengrab attack was carried out by view from the victim's perspective Light Weight DIFT analysis tool. During the analysis, we have used a virtual computer network consists of two virtual computers (one virtual computer acts as the adversary and the other acts as the victim) configured using VirtualBox (See Figure 4.1). To simulate the multi-host screengrab attack, we first executed the start and terminate script (discussed in 4.2.1) on the adversary virtual computer such that this script will start the Netlink based logger (discussed in 4.2.2) right before the execution of the attack. During the execution of the multi-host screengrab attack on the victim virtual computer, Netlink based logging program logs all the processes created during the attack time frame including process related to the attack as well as background processes. Upon completion of the attack, start and termination script sends another signal to the victim side to terminate the Netlink based logger on the victim virtual computer (Netlink based logger will finish logging processes 20 seconds after the termination signal is received). Once logging is completed, a log file named "log.text" will be saved in "/Desktop/logger" directory of the victim computer as well as a screenshot file on the victim's desktop as a result of the screengrab attack (See Figure 4.2). Then log file is fed to visualization tool (discussed in 4.2.3) to find the relationship between each process and to visualize processes using process trace graph (See Figure 4.3).

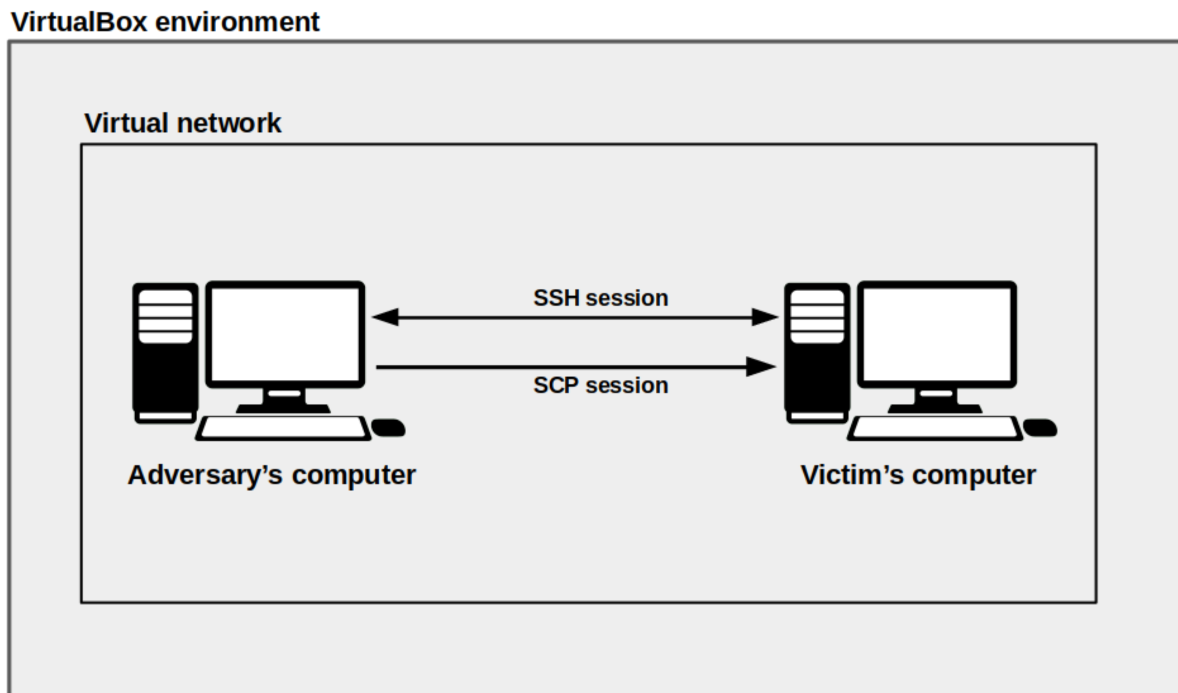


Figure 4.1: Virtual network created within VirtualBox environment connecting adversary and victim computers

Graph produced by the visualization tool mainly consists of five distinct node types:

1. SCP process node

   During the execution of the multi-host screengrab attack, SCP process node acts as the entry process of the whole attack scenario where adversary injects the files and scripts needed to perform the screengrab attack as ".zip" payload to the target computer in the network. SCP process spawns from a background process running within the Linux environment which eventually acts as the parent process to the SCP process (See Figure 4.4). SCP process was first created when the adversary initiate a SCP communication protocol with the victim computer, which will navigate to the "/tmp" directory and place the "screengrab.zip" payload which will be used by SSH processes

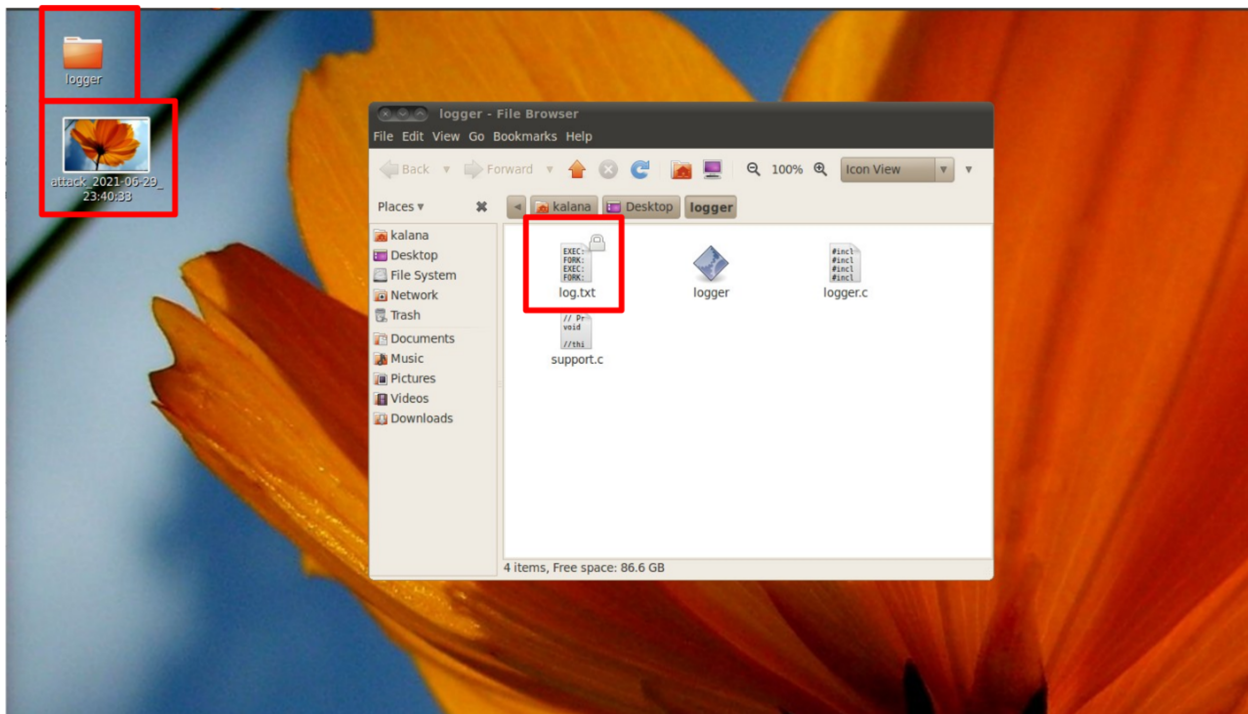


Figure 4.2: Screenshot file produced during the screengrab attack and the log file produced by the Netlink based logger

created during the attack time frame to carry out the screengrab attack.

2. SSH process nodes

During the attack execution, two SSH processes are created, where first SSH process (See Figure 4.5) with process ID 6907 spawns three children processes. One children process (process ID 6908) spawned by the first SSH parent process will change its directory to "/tmp" directory where the screengrab payload was first injected to the victim computer. The second process spawned (process ID 6912) directly from the first SSH parent process will unzip the payload by itself and spawns another children process (process ID 6913) to perform Linux package repository update via "sudo apt-get update" which will eventually spawns more children processes to update all the

existing packages installed in the Linux operating system. Once all the package were successfully updated by the process 6192 and its children, first SSH process (process ID 6907) will spawn another children process (process ID 6981) to execute "sudo apt-get install libx11-dev" and "sudo apt-get install libpng-dev" (packages needed to capture screenshots and to store them in .png format in the Linux file system respectively) by the processes 6981 and 6982 respectively.

Second SSH process (See Figure 4.6) with process ID 6788 was created to carry out the second phase of the multi-host screengrab attack. Where second SSH parent process will spawn three new children processes. The first children process (process ID 6789) will change the current working directory of the SSH session to "/tmp/screengrab" where the screengrab payload was unzipped by the process 6912 during the first SSH session. Second children process (process ID 6793) will execute the "make clean" command which will spawn more processes to scan "/tmp/screengrab" directory to find all the pre-compiled object files and then delete them in order to re-compile the code to match the victim computer's specification using victim computers C++ compiler. Finally second SSH process will create another process (process ID 6862) to execute "make" command, which will spawn another children process (process ID 6863) that will invoke victim C++ compiler to compile to the screengrab attack source code (all the children processes created by this process except for process ID 6894 are created during the screengrab source code compilation process). Upon completion of the compilation process, process 6863 was utilized by the attack to spawn process 6894 (Purple node) to execute the compiled screengrab attack executable.

3. Attack process nodes

   As discussed in subsection 4.4.1 SSH process nodes, attack process nodes are processes spawned during the execution of the multi-host screengrab attack. In the case of multi-host screengrab attack, these attack nodes can be traced back all the way to one of SSH parent process nodes. Some of the actions performed by these nodes can

be identified as updating repository packages, installing dependencies needed to carry out the attack, compiling the screengrab source code to fit the victim system etc. The amount of attack related process nodes created depends on the many factors such as operating system, Kernel version, C++ standard and C++ compiler version.

4. Background process nodes

   Background processes (See Figure 4.7) are nothing but benign process nodes captured during the time frame of the multi-host screengrab attack execution. These processes do not have any relationship to the attack process nodes. The amount of background processes captured within the time frame can drastically change depending on the current state of the Linux operating system. For instance, if the multi-host screengrab attack is taking place while the system is in idle state, the amount of background process recorded will be minimal. On the other hand, if the victim system is carrying out many CPU intensive tasks (i.e. multiple programs running at the same time or running a single multi-threaded, multi-process program) the amount of background processes captured during the attack time frame will be larger compared to when the system is in idle state.

5. Screengrab attack node

   Screengrab attack node (Purple node in Figure 4.6) is process responsible for executing the screengrab attack binary on the victim's computer once the compilation of the screengrab source code is finished by the attack process nodes spawned during the adversary's second SSH session. During the run-time of the screengrab attack, only one screengrab attack node (Purple node) is created since the source code of the screengrab attack is written single-process, single-threaded execution in mind, which allows the adversary to run this attack even on older systems.

4.4.2 Adversary-side analysis

Adversary-side analysis of multi-host screengrab attack was performed using view from adversary's perspective Light Weight DIFT analysis tool. Similarly to the victim-side analysis scenario, two virtual computers networked via virtual network within VirtualBox environment was used. However, during the adversary-side analysis Intel Pin based logging tool was directly attached to the multi-host screengrab attack's shell script to capture and log processes spawned during the execution. Once the attack successfully concluded and logging is completed by the Intel Pin based logging tool a log file named "processtrace.txt" is created in "/Desktop/Multi-host-screen-grab" directory of the adversary's computer. This log file is then fed into the visualization tool of view from the adversary's perspective Light Weight DIFT to find the relationship between each process and to visualize processes using process trace graph (See Figure 4.8).

Graph produced by the view from the adversary's perspective visualization tool consists of two distinct node types:

1. Entry process node

   This node represents the starting point of the attack. Once the attack script is launched from the Linux terminal, parent process of process 1822 (i.e. process executing the Linux terminal) will spawn a children process (i.e. process ID 1822 in Figure 4.8) to execute the attack script.

2. Attack process nodes

   Process nodes spawned by the entry process node are the attack process nodes. These attack process nodes can traced back to the entry process node where process 1867 was spawned by the process 1822 to launch an instance of sshpass[4], then sshpass instance running on process 1867 will spawn another children process (process ID

[4]sshpass is a simple and lightweight command line tool for Linux that can be directly used in SSH or SCP based bash scripts to automate the authentication process when connecting to a remote computer.

1874) to establish a SCP communication channel to the victim computer to inject the screengrab payload. Upon completion of the payload injection, process 1822 will spawn another children processs (process ID 1834) to launch another instance of the sshpass. This sshpass instance running on process 1834 will spawn another children process (process ID 1841) to establish SSH communication channel with the victim computer (first SSH session discussed in 4.4.1 bullet point 2). Once tasks related to the first SSH session is completed, another SSH session (process ID 1854) will be created (second SSH session discussed in 4.4.1 bullet point 2) to clean the pre-compiled code, re-compile the screengrab attack's source code to fit the requirements of the victim's computer and finally to remotely execute the compiled screengrab binary through the SSH session.

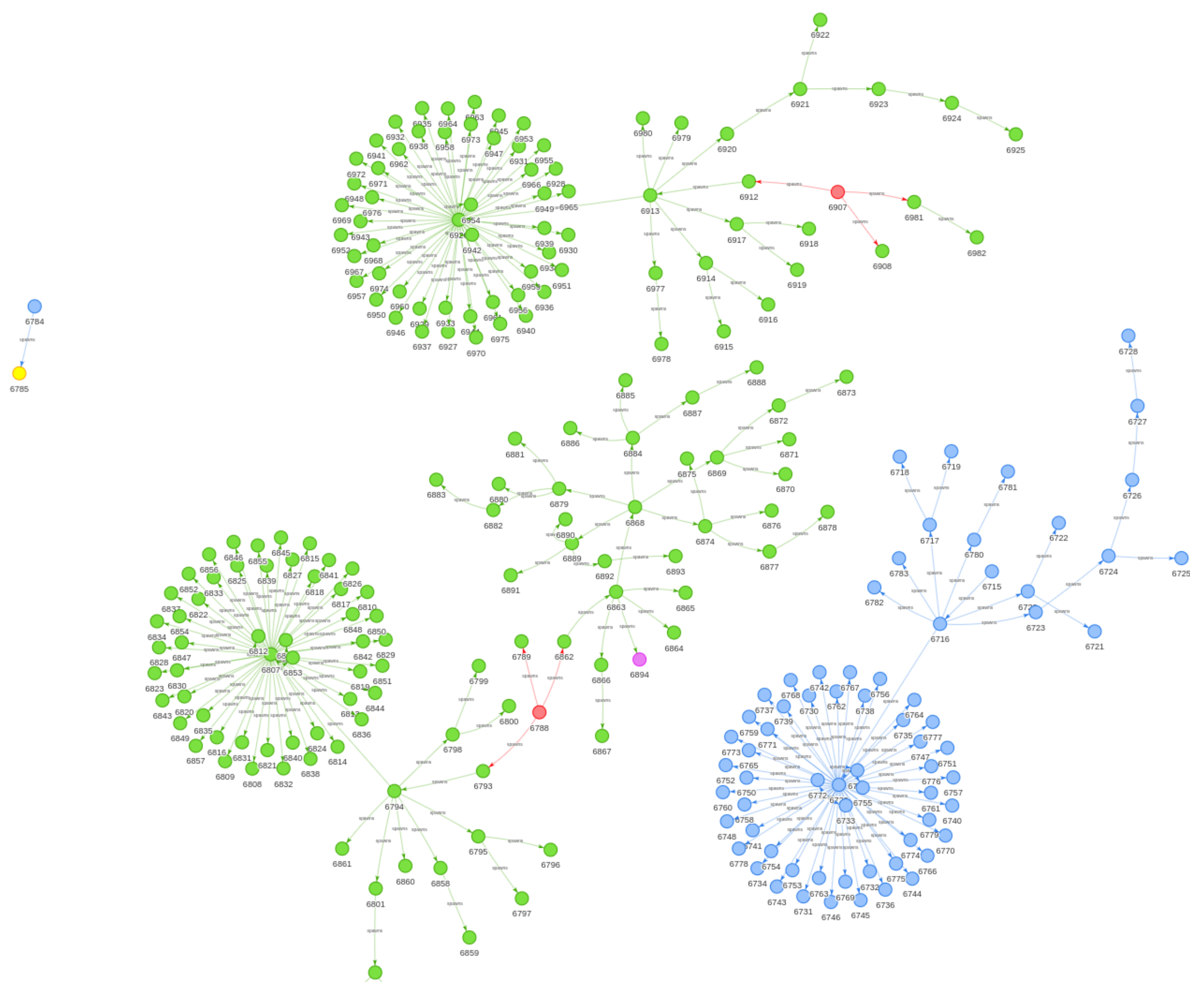


Figure 4.3: Complete process trace graph produced by the view from victim's perspective Light Weight DIFT's visualization tool

Figure 4.4: SCP process (Yellow node) created by the Linux background process (Blue node) during the execution of the multi-host screengrab attack

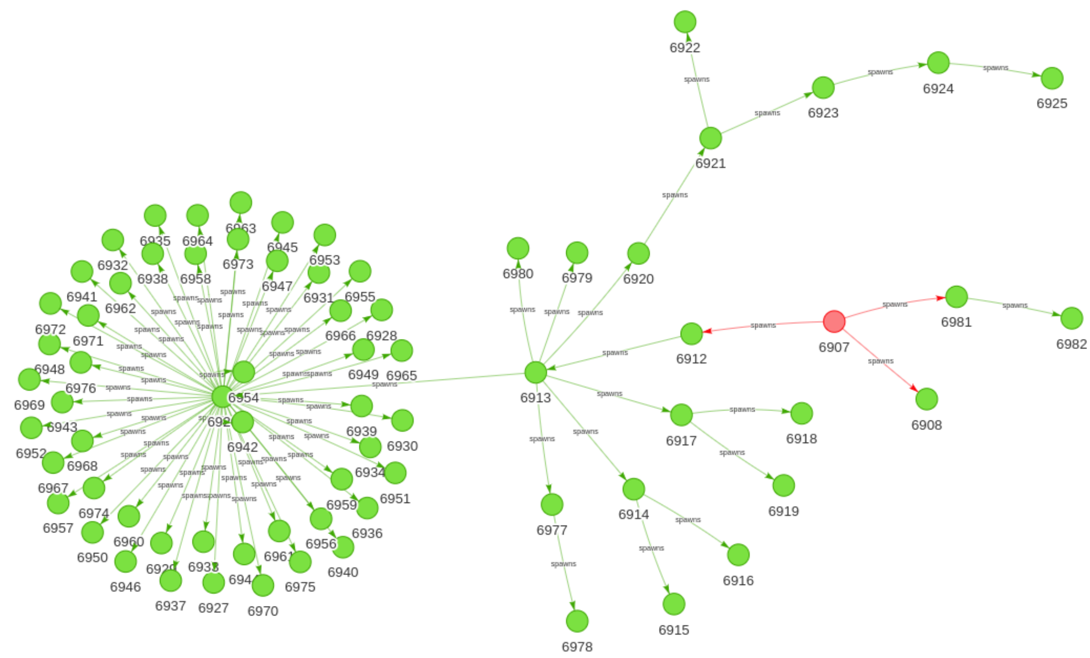


Figure 4.5: SSH process spawned by the Linux operating system (Red node) during the first SSH session from the adversary and it's children (Green nodes) created to execute commands required for the initial set-up of the attack

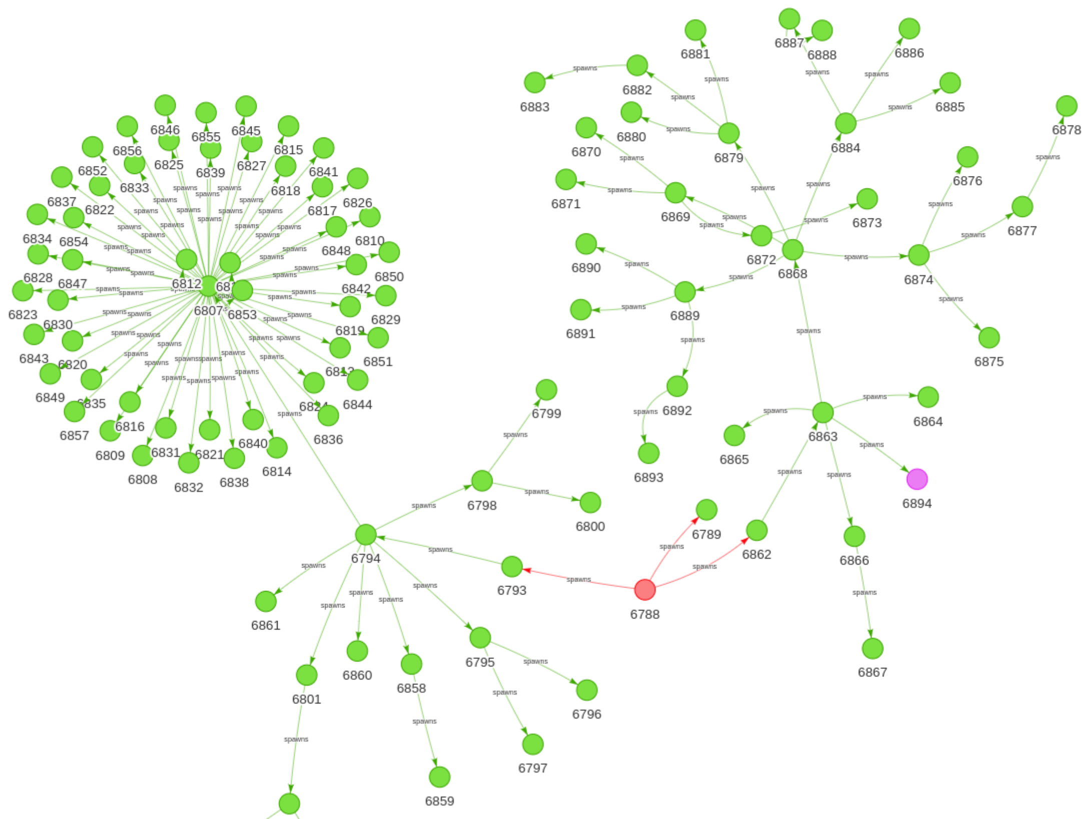


Figure 4.6: SSH process spawned by the Linux operating system (Red node) during the second SSH session from the adversary and it's children (Green nodes) created to carry out the attack

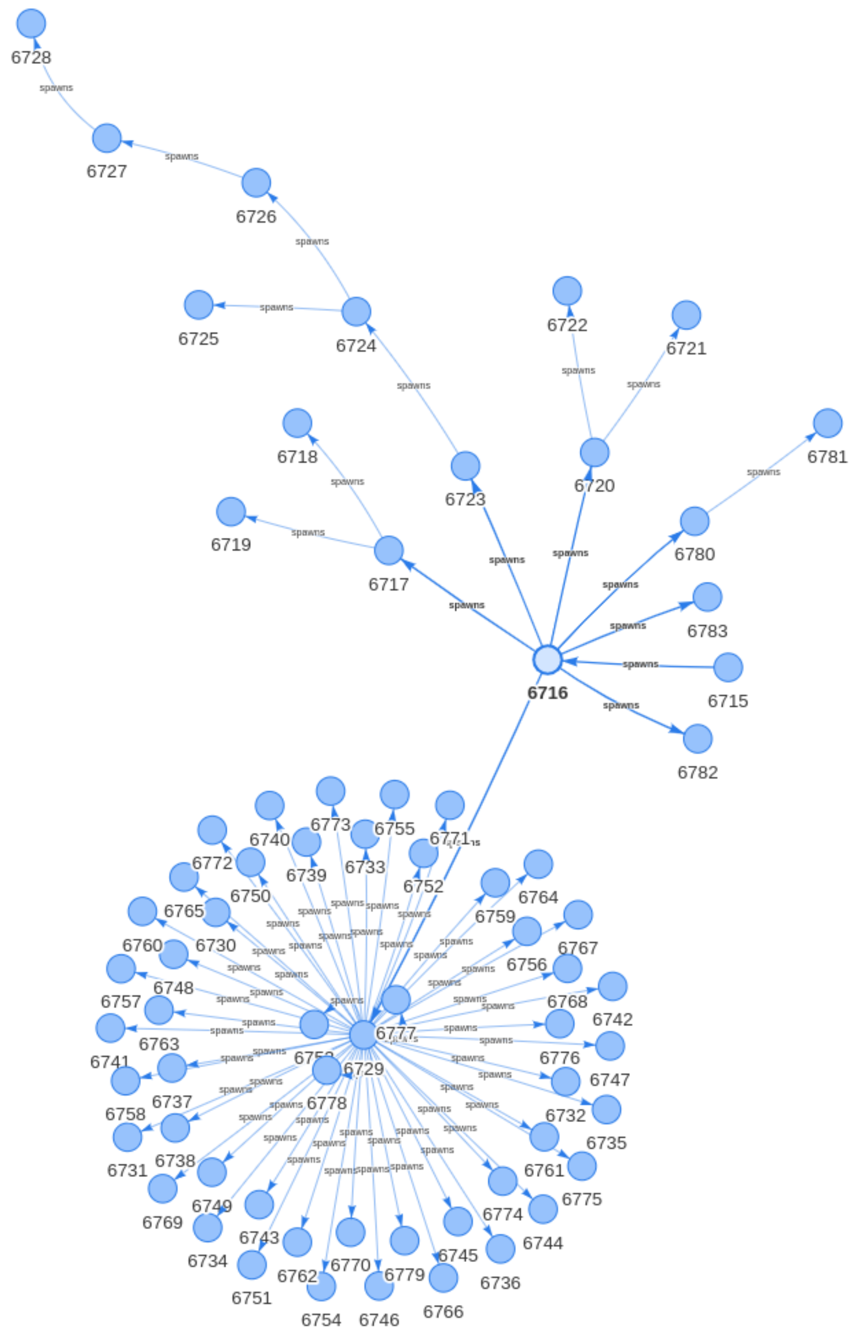


Figure 4.7: Background processes (Blue nodes) recorded during the time frame of the multi-host screengrab attack

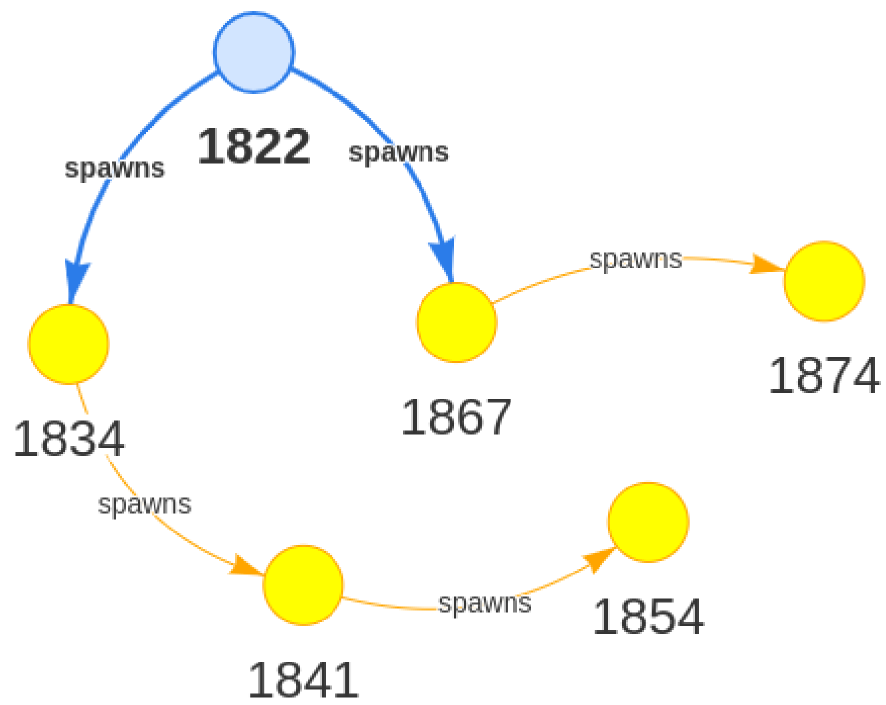


Figure 4.8: Complete process trace graph produced by the view from adversary's perspective Light Weight DIFT's visualization tool.

Chapter 5
FUTURE WORK AND CONCLUSIONS

Sections 5.1 and 5.2 of this chapter discuss the possible improvements that can be done for the proposed DIFT techniques RESTful DIFT (introcued in Chapter 2) and Light weight DIFT (introduced in Chapter 3), respectively. More specifically the modifications and extensions discussed in Sections 5.1 and 5.2 will further increase the compatibility and scalability of the proposed DIFT techniques while reducing the memory and run-time overhead. Finally, Section 5.3 presents the concluding remarks of this master's dissertation.

5.1 RESTful DIFT future work

1. Upgrade RESTful DIFT code base to support 64-bit Linux distributions.

   Current implementation of RESTful DIFT requires around 32 Terabytes of memory to enable DIFT in 64-bit systems. This is due to the LIBDFT [17] tag map structure (flat bitmaps) shared in RESTful DIFT. Updating the code base of RESTful DIFT with a new tag map structure that can perform memory efficient tagging in 64-bit systems can improve the compatibility of RESTful DIFT.

2. Develop a theory to select a set of tag sources for RESTful DIFT.

   RESTful DIFT assumes all the input channels of the DIFT enabled processes are suspicious and tags all the information flows originate at these input channels. Proposing a framework to perform tagging at only a subset of a process's input channels while maintaining high detection probabilities can reduce the tagging memory overhead incurred by RESTful DIFT. One such framework can be built by using Machine Learning (ML) to assign set of threat scores to input channels of each process. Input values for

such a ML model will require to consist of a set of features that will characterize the authenticity of each input channel. Examples of such features include but are not limited to the average payload of input channel, number of outside the system and with in the system information flows through each input channel, and average duration of a information flow at input channel. Then a threat score based optimization model can be used to find the set of best input channels to perform tagging while maintaining an acceptable detection rate.

3. Allow RESTful DIFT to handle multi-process programs

   RESTful DIFT can only handle single process programs as a consequence of the shared code base with LIBDFT. Proposing a new framework for LIBDFT code base that allows CPU parallelization of DIFT for multi-process programs can resolve this issue. Such approach can enable deployment of RESTful DIFT to protect systems from cyber threats that targets modern CPU intensive programs such as Firefox, MS word and Adobe Photoshop.

4. Modify the code base of RESTful DIFT to enable strategic security checks suggested by DIFT games research work.

   There has been research on game-theoretic modeling of DIFT to reduce the memory and run-time overhead in DIFT while maximizing the probability of detecting cyber threats [22, 28, 21]. A dynamic game model presents in [22] provides a set of system components to perform security analysis that will maximize the probability of detection while minimizing the overheads. However, [22] assumes the security check rules employed by DIFT is perfect (i.e., negligible false positives and false negatives). A stochastic game model and a reinforcement learning algorithm presents in [28] relaxes the assumption of DIFT's perfect security rules. The results of [28] inform DIFT with set of probabilistic security check locations in the victim system to achieve resource efficient DIFT while maximizing the probability of detection. A constant-sum stochastic

game model and a supervised learning-based algorithm proposed in [21] also considers imperfect security rules of DIFT and provides best security check strategies reducing the overhead.

However, the maximum effectiveness (in terms of overhead and detection probability) of the security check strategies studied in [22, 28, 21] can be achieved when DIFT is used across all the processes that are online during the system execution. Hence, it is required to first improve the code base of RESTful DIFT to handle multi-process programs (e.g., Firefox, MS-Word, MS-Excel). As the number of processes grow, it is also required to research on more memory efficient DIFT implementation that can facilitate DIFT on large number of processes. After aforementioned modifications to RESTful DIFT code base that can allow system wide DIFT, the solutions of DIFT game models in [22, 28, 21] can be implemented in DIFT as security check strategies to further reduce the overhead and promote the use of RESTful DIFT to protect large scale computer systems against more advanced cyber threats (e.g., ransomware, advanced persistent threats).

5.2 Light Weight DIFT future work

1. Proposing graph reduction methods to prune out background processes from the process trace graphs of Light Weight DIFT.

   It has been observed in Chapter 3 that the majority of the nodes and edges in the process trace graphs of Light Weight DIFT are related to the background processes. However, simply white listing background processes can lead to misdetection in the cases where cyber adversaries operate through background processes of the victim system (e.g., keylogger spyware, advanced persistent threats). Therefore, a proper graph reduction algorithm that allows pruning out background processes from the process trace graphs with minimum impact on the detection probability can provide graphs with more concise view of the security sensitive events. Maintaining a database

of prominent baseline graphs (behaviors) related to a set of background processes and comparing the background process related sub-graphs in the process trace graph in terms of a distance measure such as graph edit distance [10] may help developing such pruning algorithms.

2. Automate Light Weight DIFT-based cyber threat detection using machine learning.

   Light Weight DIFT requires a thorough analysis of its output process trace graphs by a systems security expert to decide whether a host system is under a cyber threat or not. The work load required by the systems security expert can be overwhelming when Light Weight DIFT is used to protect large scale busy computer systems as they can generate more complex and densed graphs. Furthermore, identifying the presence of more advanced cyber threats (e.g., advanced persistent threats) can require searching for correlated sequences of events in the process trace graphs which can be often exhaustive for a human expert to manually perform.

   Recent advancements in graph embedding [11] such as graph2vec [24] can enable the use of machine learning based anomaly detection algorithms [26] to determine whether the process trace graphs of Light Weight DIFT corresponding to an cyber attack or not. Figure 5.1 illustrates one such machine learning-based framework that can be used to automate Light Weight DIFT-based cyber threat detection.

5.3 Conclusion

Cyber threats impose threats to the security and privacy of sensitive information around the world in an alarming rate. Dynamic Information Flow Tracking provides a framework for detecting cyber threats via tagging, tracking and performing security checks to verify the authenticity of suspicious information flows. RESTful DIFT proposed in the Chapter 2 of this dissertation improves the code base of LIBDFT, a widely used shared libarary implementation of DIFT, to enable DIFT on multiple user defined single process programs. It also

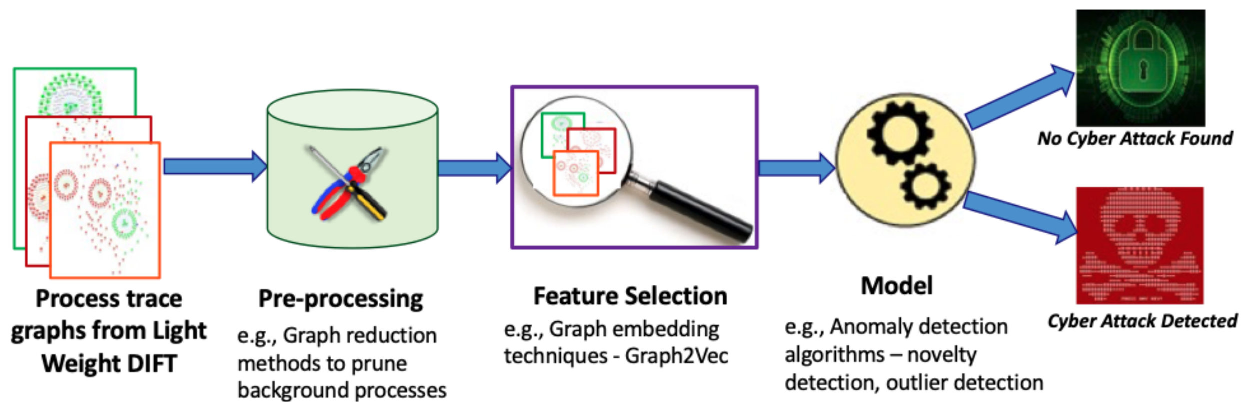


Figure 5.1: Machine learning framework for automating cyber threat detection using Light Weight DIFT.

updates the code base of LIBDFT to support the latest version of Intel Pin that enables DIFT in modern Linux distributions. Moreover, RESTful DIFT adds a RESTful service layer for improving the usability of LIBDFT and allow remote initiation and termination of DIFT on host processes. RESTful DIFT was validated through using Screengrab attack. Chapter 3 of this dissertation proposed Light weight DIFT, a low memory and run time overhead DIFT technique that supports DIFT on multi process programs and provides. Light weight DIFT can be deployed in any device and any operating system. It also consist of a adversary-side implementation that enables the behavioral study of known attack samples that can be used to derive attack signatures. Furthermore, it provides user friendly visualization of the DIFT results of the host system as a process tree graph. Light weight DIFT is validated using Multi-host Screengrab attack.

BIBLIOGRAPHY

[1] Neo4j documentation, May 2021.

[2] Catalin Cimpanu. Hackers are increasingly destroying logs to hide attacks, Nov 2018.

[3] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206, 2007.

[4] Intel Corporation. Pin 3.18 user guide.

[5] Baojiang Cui, Fuwei Wang, Tao Guo, Guowei Dong, and Bing Zhao. Flowwalker: A fast and precise off-line taint analysis framework. In *2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies*, pages 583–588. IEEE, 2013.

[6] Alec T Dean. *The Growth of Ransomware and Its Impact on City Governments*. PhD thesis, Utica College, 2019.

[7] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.

[8] Roy Thomas Fielding. Rest: Architectural styles and the design of network-based software architectures. *Doctoral dissertation, University of California*, 2000.

[9] Prahlad Fogla, Monirul I Sharif, Roberto Perdisci, Oleg M Kolesnikov, and Wenke Lee. Polymorphic blending attacks. In *USENIX security symposium*, pages 241–256, 2006.

[10] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. A survey of graph edit distance. *Pattern Analysis and applications*, 13(1):113–129, 2010.

[11] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94, 2018.

[12] Keman Huang, Rebecca Ye, and Stuart Madnick. Both sides of the coin: The impact of cyber attacks on business value. 2019.

[13] Eric Jardine. Taking the growth of the internet seriously when measuring cybersecurity. *Researching Internet Governance: Methods, Frameworks, Futures*, 2020.

[14] Kangkook Jee, Georgios Portokalidis, Vasileios P Kemerlis, Soumyadeep Ghosh, David I August, and Angelos D Keromytis. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *NDSS*, 2012.

[15] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 377–390, 2017.

[16] Hari Kannan, Michael Dalton, and Christos Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 105–114. IEEE, 2009.

[17] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 121–132, 2012.

[18] Kenneth Kimani, Vitalice Oduol, and Kibet Langat. Cyber security challenges for iot-based smart grid networks. *International Journal of Critical Infrastructure Protection*, 25:36–49, 2019.

[19] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, Ramachandran Sekar, and VN Venkatakrishnan. Holmes: Real-time apt detection through correlation of suspicious information flows. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1137–1152. IEEE, 2019.

[20] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. Straighttaint: Decoupled offline symbolic taint analysis. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 308–319. IEEE, 2016.

[21] Shana Moothedath, Dinuka Sahabandu, Joey Allen, Linda Bushnell, Wenke Lee, and Radha Poovendran. Stochastic dynamic information flow tracking game using supervised learning for detecting advanced persistent threats. *arXiv preprint arXiv:2007.12327*, 2020.

[22] Shana Moothedath, Dinuka Sahabandu, Joey Allen, Andrew Clark, Linda Bushnell, Wenke Lee, and Radha Poovendran. A game-theoretic approach for dynamic information flow tracking to detect multistage advanced persistent threats. *IEEE Transactions on Automatic Control*, 65(12):5248–5263, 2020.

[23] Steve Morgan. Cybercrime to cost the world $10.5 trillion annually by 2025. *Cybercrime Magazine*, 13, 2020.

[24] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. graph2vec: Learning distributed representations of graphs. *arXiv preprint arXiv:1707.05005*, 2017.

[25] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

[26] Salima Omar, Asri Ngadi, and Hamid H Jebur. Machine learning techniques for anomaly detection: an overview. *International Journal of Computer Applications*, 79(2), 2013.

[27] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 135–148. IEEE, 2006.

[28] Dinuka Sahabandu, Shana Moothedath, Joey Allen, Linda Bushnell, Wenke Lee, and Radha Poovendran. A multi-agent reinforcement learning approach for dynamic information flow tracking games for advanced persistent threats. *arXiv preprint arXiv:2007.00076*, 2020.

[29] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *ACM Sigplan Notices*, 39(11):85–96, 2004.

[30] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 255–264, 2002.

[31] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127, 2007.

Appendix A

SOURCE CODE OF THE SCREENGRAB

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <malloc.h>
4 #include <X11/Xlib.h>
5 #include <X11/X.h>
6 #include <png.h>
7 #include <errno.h>
8 #include <time.h>
9
10 int main(int argc, char *argv[])
11 {
12  const char* title = NULL;
13  XWindowAttributes gwa;
14  int result = 0;
15  FILE *fp = NULL;
16  int width, height;
17  int x, y;
18  png_structp png_ptr = NULL;
19  png_infop info_ptr = NULL;
20  png_bytep row = NULL;
21  if (argc < 2)
22  {
23   fprintf(stderr, "usage: %s <outputFileName> <optionalTitle>\n", argv[0]);
24   return 1;
25  }
26  char* filename = argv[1];
27  if (argc == 3)
```

```
28    title = argv[2];
29   else
30    title = "screengrab";
31
32   Display *display = XOpenDisplay(":0.0");
33   if (display == NULL)
34   {
35    fprintf(stderr, "couldn't open display\n");
36    return 1;
37   }
38   Window root = DefaultRootWindow(display);
39
40   XGetWindowAttributes(display, root, &gwa);
41   width = gwa.width;
42   height = gwa.height;
43
44   XImage *image = XGetImage(display,root, 0,0 , width,height,AllPlanes, ZPixmap
       );
45
46   char file_name_with_time_stamp[200];
47   time_t t = time(NULL);
48   struct tm time_date = *localtime(&t);
49   snprintf(file_name_with_time_stamp, sizeof(file_name_with_time_stamp), "%s_%d
       -%02d-%02d_%02d:%02d:%02d",filename,time_date.tm_year + 1900, time_date.
       tm_mon, time_date.tm_mday, time_date.tm_hour, time_date.tm_min, time_date.
       tm_sec);
50   fp = fopen(file_name_with_time_stamp, "wb");
51   if (fp == NULL)
52   {
53    fprintf(stderr, "Could not open file %s for writing\n",
        file_name_with_time_stamp);
54    result = 1;
55    goto finish;
```

```
56   }
57
58   png_ptr = png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
59   if (png_ptr == NULL) {
60    fprintf(stderr, "Error creating png\n");
61    result = 1;
62    goto finish;
63   }
64
65   // Initialize info structure
66   info_ptr = png_create_info_struct(png_ptr);
67   if (info_ptr == NULL) {
68    fprintf(stderr, "Error during png creation\n");
69    result = 1;
70    goto finish;
71   }
72
73   // Setup Exception handling
74   if (setjmp(png_jmpbuf(png_ptr))) {
75    fprintf(stderr, "Error during png creation\n");
76    result = 1;
77    goto finish;
78   }
79
80   png_init_io(png_ptr, fp);
81
82   png_set_IHDR(png_ptr, info_ptr, width, height,
83      8, PNG_COLOR_TYPE_RGB, PNG_INTERLACE_NONE,
84     PNG_COMPRESSION_TYPE_BASE, PNG_FILTER_TYPE_BASE);
85
86   png_text title_text;
87   title_text.compression = PNG_TEXT_COMPRESSION_NONE;
88   title_text.key = (png_charp)"Title";
```

```
89    title_text.text = (png_charp)title;
90    png_set_text(png_ptr, info_ptr, &title_text, 1);
91
92    png_write_info(png_ptr, info_ptr);
93
94    row = (png_bytep) malloc(3 * width * sizeof(png_byte));
95
96    for (y=0 ; y<height ; y++)
97    {
98     for (x=0 ; x<width ; x++)
99     {
100     unsigned long pixel = XGetPixel(image,x,y);
101     (&row[x*3])[0] = (pixel & image->red_mask   ) >> 16;
102     (&row[x*3])[1] = (pixel & image->green_mask ) >> 8;
103     (&row[x*3])[2] = (pixel & image->blue_mask  );
104    }
105    png_write_row(png_ptr, row);
106   }
107
108   // End write
109   png_write_end(png_ptr, NULL);
110
111  finish:
112   if (fp != NULL)
113    fclose(fp);
114   if (info_ptr != NULL)
115    png_free_data(png_ptr, info_ptr, PNG_FREE_ALL, -1);
116   if (png_ptr != NULL)
117    png_destroy_write_struct(&png_ptr, (png_infopp)NULL);
118   if (row != NULL)
119    free(row);
120   return result;
121  }
```