PROCESS FORENSICS: THE CROSSROADS OF CHECKPOINTING AND INTRUSION DETECTION

By

MARK FOSTER

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2004

ACKNOWLEDGMENTS

I would like to thank my supervisory committee chair Joseph N. Wilson. Without his patience and support this work would not have been possible. Many thanks are also extended to the supervisor of my teaching assistantship, Rory DeSimone. She has created a sense of home and family I will never be able to forget. Without this support I would not have continued my graduate work beyond the master's level. I would also like to thank my parents for their support. They have provided me with an amazingly reliable safety net that allowed me to focus solely on school and avoid many headaches. Lastly, and most of all I would like to thank my wife. Her love, support, and patience have not wavered in this lengthy process. She has undoubtedly been the single most integral component to my success. I am proud to say that her days of being the bread winner are numbered.

TABLE OF CONTENTS

page

LIST OF TABLES

Table page

LIST OF FIGURES

Figure page

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

PROCESS FORENSICS: THE CROSSROADS OF
CHECKPOINTING AND INTRUSION DETECTION

By

Mark Foster

December 2004

Chair: Joseph N. Wilson
Major Department: Computer and Information Sciences and Engineering

The goal of our study was to introduce a new area of computer forensics we call process forensics. Process forensics involves extracting information from a process' address space for the purpose of finding digital evidence pertaining to a computer crime. The challenge of this subfield is that the address space of a given process is usually lost long before the forensic investigator is analyzing the hard disk and file system of a computer.

Our study began with an in-depth look at checkpointing techniques. After surveying the literature and developing our own checkpointing tool, we believe checkpointing technology is the most appropriate method for extracting information from a process's address space. We make the case that an accurate and reliable checkpointing tool provides a new source of evidence for the forensic investigator. We also thoroughly examined the literature and methods for detecting buffer overflow attacks. In addition, we have developed a new method for detecting the most common form of a buffer

overflow attack, namely, stack-smashing attacks. We believe that the boundary where these two areas meet (specifically, incorporating checkpointing with intrusion detection) can readily provide process forensics. The technology of checkpointing is nothing new when considering process migration, fault tolerance, or load balancing. Furthermore, a plethora of research has already focused on finding methods for detecting buffer overflow attacks. However, with respect to computer forensics, the gains from incorporating checkpointing with intrusion detection systems have yet to be explored.

CHAPTER 1
INTRODUCTION

In recent years, computers and the Internet have become an integral part of our society. Computer use includes the workplace, home, school, and in some cases, public areas such as shopping malls and airports. The National Telecommunications and Information Administration (NTIA) released a report showing that Internet growth in the United States is estimated at 2 million new users each month [1]. The downside to this trend of pervasive computing is that the amount of computer-based crime is also on the rise. Statistics published by the CERT Coordination Center [2] show that the number of security related incidents have increased every year since 1998. From 2001 to 2003, the number of security related incidents more than doubled. As computer crime increases, so do the demands placed on computer security specialists and law enforcement.

To many computer security specialists, intrusion prevention is more important than intrusion detection. However, as long as intruders continue to be successful, the need for reliable intrusion detection systems is apparent. In addition, to prevent repetitive or similar intrusive attacks, we need reliable computer forensics to help us learn why an attack occurred in the first place. Thus, computer forensics is an integral part of intrusion prevention.

Our purpose is to introduce a new area of computer forensics, called *process forensics*. Process forensics involves extracting information from the process address space of a given program. We discuss how the information extracted from a process address space could be a source of evidence after a computer crime. However, to collect

digital evidence in the form of process forensics, one must address two issues. First, some tool must exist that can extract information from a process address space. Second, one must know when to extract such information. We propose that checkpointing technology be used for the extraction process. Checkpointing research is already aimed at storing key information about a process. We believe that the proper checkpointing tool can also meet the needs of the evidence collector. Furthermore, we propose that intrusion detection systems be used to help indicate when to extract information from a process address space. Intrusion detection systems already aim to detect malicious activity. Malicious activity sometimes results in the need for evidence in a computer crime.

Checkpointing

Individuals familiar with UNIX systems are probably also familiar with the act of stopping and restarting a running process. This act is usually achieved by sending a series of signals to a given process. The signals SIGSTOP and SIGCONT are used for this purpose. While stopping and restarting processes is a useful capability, it is important to note that during the time a process is stopped, it still consumes system resources. In particular, a stopped process consumes system memory associated with saving its state. Without saving the process's state, we cannot restart the process. Checkpointing is a way of saving a process's state such that the process may be restarted from the point at which it was checkpointed without requiring active operating system information. Checkpoints are made at regular or chosen time intervals. Checkpointing plays a vital role in fault tolerance and rollback recovery schemes. After a system crash or failure, processes that were checkpointed can be restarted from the point of their last checkpoint.

We introduce a new system for checkpointing called UCLiK (Unconstrained Checkpointing in the LInux Kernel). UCLiK has been implemented as a kernel module for the Linux operating system. UCLiK is different from most other checkpointing systems in that it requires no additional programming by the application programmer. Furthermore, it requires no special compiler or run-time library. UCLiK operates on the system-level, and therefore has direct access to all of a process's state. When UCLiK checkpoints a process, it does so by taking a snapshot of a process's state and saving that information to a file. By saving this state to a file, we no longer use system resources, except for whatever stable storage is necessary for storing a file. We refer to the file containing our saved state as our *image* file. The image file can be stored indefinitely, or moved to another system where the process could be restarted to achieve process migration.

Some well-known benefits of checkpointing include process migration, fault tolerance, and rollback recovery. The benefits of our system are even more widespread. System administrators can use such a system in place of killing what are seemingly objectionable processes. Killing what seems to be a problem process, but in actuality is not, can result in a drastic loss of computation for the user and the system. The user is upset at having to start the process over again, and the system repeats computation it has already performed. With UCLiK, such a process could be checkpointed to a file and later restarted if the administrator can determine that running the process presents no problem. Essentially, UCLiK can serve as an undo option for the kill system call. Brown and Patterson [3] made a thorough case for the importance of a system-level undo mechanism. Furthermore, they showed that human errors are inevitable and should be

considered when designing highly-available and highly-dependable systems [4]. A common example of human error believed to be experienced by many including this author when issuing the kill system call is using the wrong process identification (PID). Obviously this results in killing the wrong process. With UCLiK, one can undo such a mistake. With UCLiK, a system administrator can be more trigger-happy when killing suspicious user processes. Rather than waiting until a user process is blatantly degrading the system's performance, system administrators can take a more preemptive approach and checkpoint a user's process at the first sign of trouble. UCLiK could also be used by system administrators when doing preventive maintenance. Maintenance to a system often requires taking the system down and thus killing a number of user processes. These processes could be checkpointed instead of killed. Once maintenance is complete, the user processes could be restarted.

By alleviating the need for a special compiler, a run-time library, or additional work for the application programmer, we come one step closer to the ideal checkpointing system. The ideal checkpointing system would allow us to checkpoint any process, anytime, anywhere. Furthermore, one should be able to restart a checkpointed process anytime and anywhere. Ultimately, the ideal operating system would have the ability to checkpoint and restart running processes.

Buffer Overflow Attacks

The term *buffer overflow* refers to copying more data into a buffer than the buffer was designed to hold. A *buffer overflow attack* occurs when a malicious individual purposely overflows a buffer to alter a program's intended behavior. In most common forms of this attack, the attacker intentionally overflows a buffer on the stack so that the excess data overwrites the return address just below the buffer on the stack. When the

current function returns, control flow is transferred to an address chosen by the attacker. Commonly, this address is a location on the stack, inside the buffer, where the attacker has injected malicious code. This type of buffer overflow attack is also referred to as a stack-smashing attack, since the buffer resides on the stack. Stack-smashing attacks are among the most common buffer overflow attacks because of their simplicity of implementation.

Buffer overflow attacks have been a major security issue for years. Wagner et al. [5] extracting statistics from CERT advisories, found that between 1988 and 1999, buffer overflows accounted for up to 50% of the vulnerabilities reported by CERT. Other statistics showed that buffer overflows caused at least 23% of the vulnerabilities in different databases. More recent National Institute of Standards and Technology's (NIST) ICAT statistics show that a significant number of the common vulnerabilities and exposures (CVE) and CVE candidate vulnerabilities were due to buffer overflows. For the years 2001, 2002, and 2003, buffer overflows accounted for 21, 22, and 23% of the vulnerabilities respectively [6]. From April 2001 to March 2002, buffer overflows caused 20% of the vulnerabilities reported by SecurityTracker [7]. These more recent statistics reinforce the case made by Wagner et al. [5]. Buffer overflows are a significant issue for system security.

We introduce a new method for detecting stack-smashing and buffer overflow attacks. While much work has been focused on detecting stack-smashing attacks, few approaches use the program call stack to detect such attacks. Our new method of detecting stack-smashing attacks relies solely on intercepting system calls and information that can be extracted from the program call stack and process image. Upon

intercepting a system call, our method traces the program call stack to extract return addresses. These return addresses are used to extract what we refer to as *invoked addresses*. In the process image, return addresses are preceded by *call* instructions. These *call* instructions are what placed the return addresses on the stack and then transferred control flow to another location. An address that was invoked by a *call* instruction is referred to as an *invoked address*. We use the return and invoked addresses to create a weighted directed graph. We found that the graph constructed from an uncompromised process always contains a Greedy Hamiltonian Call Path (GHCP). This allows us to use the lack of a GHCP to indicate the presence of a buffer overflow or stack-smashing attack.

CHAPTER 2
BACKGROUND AND RELATED WORK

Checkpointing

Background

Checkpointing is the technique of storing a running process's state in such a way that a process can be restarted from the point at which the checkpoint, or stored copy of a process state, was created. Checkpointing plays an important role in fault tolerance, rollback recovery, and process migration. After a system crash, processes that have been checkpointed will be able to restart from their most recent checkpoints, while uncheckpointed processes have lost their work and will be forced to start over from the beginning. Process migration is the act of moving a running process from one host to another. Process migration can be achieved by checkpointing a running process on one host, moving the checkpoint to another host, and then restarting the process on the new host.

The amount of stable storage required to save a checkpoint is called the *checkpoint size*. The amount of time it takes to make a checkpoint is referred to as the *checkpoint time*. The additional amount of time it takes to run an application when checkpointing that application as opposed to when that application is not being checkpointed is referred to as the *checkpoint overhead* [8].

Most checkpointing techniques can be classified into two major categories: user-level and system-level. User-level checkpointing refers to checkpointing systems whose code executes on the user-level. User-level checkpointing systems execute in user-space.

Checkpointing at the user-level usually involves an enhanced language and compiler, a run-time library with checkpointing support, or modifications to the source code of a process to be checkpointed. System-level checkpointing refers to checkpointing systems whose code executes as part of the operating system. These systems are also referred to as kernel-level checkpointing systems. These systems execute in kernel-space. These systems usually involve modifications to an existing operating system. Debate is ongoing as to whether checkpointing should take place at the user-level or system-level.

The ultimate goals of a checkpointing system are transparency, portability, and flexibility. The ideal checkpointing system is completely transparent to the application programmer. Ideally, the application programmer can write code however he wants, while using any language and any compiler, and still be able to have his application checkpointed. Furthermore, this checkpointing system should be portable to any type of system. In addition, the checkpoint of a process should be portable to any type of system for a restart. All along, the user should have the flexibility to minimize the checkpoint size, time, and overhead. These aspects of checkpointing consume storage and computation.

A common method for checkpointing is to suspend the execution of a process, write the process's state information to stable storage and then continue with the execution of the process. This method of checkpointing is called *sequential checkpointing* [8]. Another checkpointing method is *forked checkpointing* [8]. With forked checkpointing, the process to be checkpointed is forked, and the parent process continues with execution, while the child process is used for making the checkpoint. For long-running processes, multiple checkpoints may be taken. *Incremental checkpointing*

refers to storing only what has changed in the process's state since the previous checkpoint [8]. All other information can be extracted from the previous checkpoints. Checkpoints can be made *synchronously* or *asynchronously* [8]. When the application programmer specifies points in the code at which to perform checkpoints, these checkpoints are said to be synchronous. If a checkpoint is taken at regular time periods (such as every hour for a long-running process), it is said to be asynchronous.

One common approach to reducing checkpoint size is *memory exclusion* [8]. The idea behind memory exclusion is that clean and dead areas of memory need not be included into a checkpoint. Clean areas of memory are portions of memory that have not changed since the previous checkpoint. Clean memory is also called Read-only memory [9]. Dead memory includes those portions of memory that will not be read before being written after the current checkpoint, and thus need not be included in the checkpoint.

Often, one may need to checkpoint a group of parallel applications or distributed processes. Usually this type of computation takes place on a network or a system of clustered computers. This scenario is a bit more complex than the sequential checkpointing of a single process. The challenge here is interprocess communication. If one node fails, and a process running on that node must rollback to its last checkpoint, any messages that a process has sent since its last checkpoint are sent again. Therefore, any other process in the system that has received a message from that process since its last checkpoint must also rollback. The rollback of one process might invoke the rollback of another, and in turn invoke the rollback of each process in the system. This concept (and the idea that rollbacks could propagate through all the processes in the system and return the system to its initial state) is called the *domino effect* [10]. To avoid

the domino effect, checkpointing of parallel applications and distributed processes usually involves some sort of *coordinated checkpointing*. Coordinated checkpointing refers to a group of processes coordinating their checkpoints to achieve a *global consistent state* [10]. A global consistent state refers to a point in the execution of a group of processes where, for any process whose state includes having received a message, there is another process whose state includes having sent that message [10]. Uncoordinated checkpointing can lead to inconsistent states, and thus lead to the domino effect.

Another technique that assists in rollback recovery of parallel applications and distributed processes is logging. Some checkpointing systems log pertinent information and events to assist in the rollback recovery of a process or group of processes. One common approach to this is to model message receipts as nondeterministic events [10]. Each nondeterministic event then has its own corresponding deterministic time interval. A nondeterministic event corresponds to the deterministic time interval after it and before the next nondeterministic event. The main objective behind this is to avoid *orphan* processes. Orphan processes are processes that are dependent on an event that cannot be generated during recovery [10].

We can now classify logging approaches into three categories: *pessimistic* logging, *optimistic* logging, and *causal* logging. With pessimistic logging, the determinant of each nondeterministic event is logged to stable storage before any other processing is done. Pessimistic logging does not allow any process to depend on an event before that event has been logged [10]. Pessimistic logging never creates any orphan processes, but imposes more overhead because of its blocking nature. With optimistic logging, the

determinant of each nondeterministic event is immediately logged to volatile storage; but may not be logged to stable storage until sometime later depending on the protocol [10]. If a failure occurs before the volatile storage is transferred to stable storage, then orphan processes may exist during a recovery. If this is the case, each orphan process is forced to rollback until it is no longer dependent on an event that cannot be generated. The benefit of this approach is a reduction in overhead. Causal logging requires logging all determinants corresponding to the nondeterministic events that are causally related to the process's given state [10]. Causal logging applies Lamport's happened-before relationship to the determinants of the nondeterministic events. This approach does not allow orphan processes. Unlike pessimistic logging, causal logging does not suffer from increased overhead.

Related Work

User-level checkpointing

Application programmer-defined checkpointing. Over the years, there have been a number of different checkpointing schemes. One straightforward checkpointing scheme has been to leave the responsibility of checkpoints to the application programmer. After all, the application programmer should know more about the code than anyone else. The application programmer should know exactly what information is pertinent and which locations are best for checkpoints. However, without any support, the task of writing fault-tolerant applications can be challenging. Some studies show that fault-tolerant routines can take up to 50% of the source code [11]. Fortunately, other approaches to checkpointing have been developed, to relieve the application programmer from such a burden.

Checkpointing with library support. To alleviate much of the burden of checkpointing placed on the application programmer, some programmers use a run-time library with support for checkpointing. One benefit of this approach is that we still exploit the application programmer's knowledge of the code. The application programmer still decides what data is checkpointed, and where the checkpoints should be placed. This is referred to as user-defined checkpointing [11]. One major advantage of user-defined checkpointing is its ability to greatly reduce the checkpoint size. Much of the process state that is saved in a system-level checkpointing scheme is able to be eliminated from the checkpoint, since the application programmer is familiar with the data involved in a process. The application programmer can decide what data is truly necessary for checkpointing the given process.

CHK-LIB is one such run-time library [11]. This run-time library provides three fault-tolerant primitives. One such primitive is used for the application programmer to specify which data should be saved in a checkpoint. Another such primitive is used to allow the programmer to specify where checkpoints should be made. The last primitive is used to determine whether a process is new or has been restarted.

Another system that uses library support for checkpointing is Condor [12]. Condor is a distributing processing system whose main goal is to maximize resource utilization by scheduling processes on idle workstations. Checkpointing and process migration play a vital role in Condor. If the owner of a workstation being used by the Condor system needs to use that workstation, Condor must be able to checkpoint any process on that workstation and migrate those processes elsewhere. Processes to be run in the Condor system are relinked with the Condor checkpointing library. This checkpointing library

contains system call wrappers that can log information such as names of opened files, file descriptor numbers, and file access modes.

Checkpoints in the Condor system are invoked by a signal. A checkpoint in the Condor system is created by copying the process's state information to a file. Restart is achieved by having the restarting process copy the original process's state information from that checkpoint file into the process address space of the restarting process. Routines for handling this signal, and for writing a process's state information into a file, are provided in the Condor checkpointing library. Process migration is achieved when a checkpoint file is moved to another location and then restarted. Condor has a number of benefits. Condor was designed to work on UNIX systems, but does not require any modifications to the UNIX kernel. Furthermore, since this is a user-level checkpointing system, the system may be more portable than a system-level checkpointing system.

Condor has been used in conjunction with other systems such as CoCheck [13]. CoCheck is system for checkpointing parallel application on networks of workstations. CoCheck is concerned primarily with finding a global consistent state for all of the nodes in a network. Once this consistent state is achieved, Condor aids in checkpointing the parallel applications. Once checkpointed, these applications can be migrated to achieve a more balanced load across a network.

Condor does suffer from a number of drawbacks. Condor makes no attempt to deal with pipes, sockets, or any other form of interprocess communication. Condor also assumes that files opened by a process being checkpointed remain unchanged when the restart takes place. Furthermore, Condor relies on a stub process running in the location of the original process to facilitate file access at the original process's location.

Checkpointing with Condor also requires that one know if a process may be checkpointed before starting that process. A process that is running and was not relinked with the checkpointing library can't be checkpointed.

Libckpt [8] is another checkpointing library designed for UNIX systems. Somewhat similar to Condor, checkpointing with Libckpt takes place at the user-level. However, Libckpt differs from Condor in that processes to be checkpointed with Libckpt must be recompiled, not just relinked with the Libckpt checkpointing library. Furthermore, Libckpt requires that the *main()* function in a program to be checkpointed, must be renamed to *ckpt_target()*. Libckpt also distinguishes itself from Condor by its additional performance optimizations. Libckpt supports incremental checkpointing and forked checkpointing. Libckpt also can do memory exclusion via user-directives. Memory exclusion with these user-directives is called user-directed checkpointing. The Libckpt library supplies two procedure calls for memory exclusion: *include_bytes()* and *exclude_bytes()*. These two procedure calls tell the system which portions of memory to include or exclude in the next checkpoint. In some cases, the checkpoint size was reduced by as much as 90% when using the user-directives to do memory exclusion [8]. The Libckpt library also supports the procedure call: *checkpoint_here()*. This procedure call enables the programmer to request a synchronous checkpoint at any point in the code. By default, Libckpt checkpoints a process every 10 minutes. The time interval between checkpoints, whether to use incremental or forked checkpointing, and a number of other options are specified in the *.ckptrc* file.

Libckpt does have a number of disadvantages. As with Condor, one must know before compilation that a process needs to be checkpointed. Transparency is lost by

requiring the programmer to rename the *main()* function. Furthermore, to achieve the performance optimizations of memory exclusion and synchronous checkpoints, additional transparency is lost by requiring the programmer to use the user-directives. Pipes and sockets were not addressed in the literature discussing Libckpt.

Compiler-defined checkpointing. Another common approach to checkpointing at the user-level is referred to as compiler-defined checkpointing. This approach consists of using the compiler's knowledge of the program to select the most optimal locations for inserting checkpoints. One advantage of this approach is that the application programmer is not responsible for checkpoints. This results in a checkpointing scheme that is transparent to the application programmer. Another advantage of this approach is that when compared with system-level checkpointing, the compiler-defined checkpoints are typically much smaller. System-level checkpointing usually requires saving most of a process's address space, also referred to as its *memory footprint*. This usually results in a large checkpoint file. However, with the recent reduction in costs for stable storage, one could argue that for larger checkpoint files are often acceptable. For example, a system administrator might rather use a few megabytes of storage than deal with a disgruntled user whose process was killed. Furthermore, some proponents of compiler-based checkpointing think that for processes with small memory footprints, system-level checkpointing is still best [14]. However, the same individuals think that compiler-defined checkpointing is better for applications on large cluster computer systems [14].

One approach to checkpointing on large cluster computer systems was to have the ZPL language and compiler perform automatic checkpointing [14]. A major challenge of checkpointing processes across clustered computer systems is the difficulty in identifying

a globally consistent state. This approach used the compiler's knowledge of the code to identify ranges of code that were void of communication. These ranges of code could be checkpointed without losing messages in the network. The compiler further exploited its knowledge of the code to insert checkpoints where the fewest live array variables existed. In some cases, these compiler techniques resulted in as much as a 73% reduction in the checkpoint size [14].

Additional compiler-based memory-exclusion techniques have used Libckpt [15]. The previously mentioned Libckpt [8] library used user-directives such as include_bytes(), exclude_bytes, and checkpoint_here() to perform the checkpointing. Libckpt was later expanded to include additional directives such as EXCLUDE_HERE and CHECKPOINT_HERE. These directives tell the compiler when to invoke the checkpoint_here() or memory-exclusion procedure calls. Having the compiler invoke these procedure calls allows the compiler to guarantee a correct checkpoint. The compiler determines what portions of memory to exclude, based on a set of data flow equations. These equations are solved using an iterative method. This differs from a careless programmer who might exclude portions of memory that are essential to recovery. However, the programmer is still responsible for inserting these new directives. This results in a continued loss of transparency. This technique is called Compiler Assisted Memory Exclusion (CAME) [15].

Compiler-defined checkpointing schemes do suffer from a number of disadvantages. One obvious disadvantage is transparency. Only programs that are compiled with compilers possessing support for checkpointing will be able to be checkpointed. Furthermore, the compiler will have no knowledge of any message-

passing that takes place during execution. To work around this disadvantage, one could log the activity across communication channels, but this obviously results in additional work and overhead. As we have discussed, some compiler defined checkpointing schemes just limit the location of checkpoints to specific ranges in the program's execution [14].

Exportable kernel state. Another interesting approach to checkpointing involves exporting the kernel state. This approach was explored using the Fluke microkernel [16]. The Fluke microkernel was designed in such a way that the kernel state is exportable to the user-level. The Fluke microkernel also allows the kernel state to be set, or imported from the user-level. This allows a user-level checkpointing system to have access to the necessary kernel objects that pertain to a given process. Most of the checkpointing systems we discussed so far must infer the kernel state by logging information collected by system call wrappers.

One of the major drawbacks of this approach to checkpointing is that a process to be checkpointed must be in the child environment of the checkpointing application. A process with no checkpointing application as an ancestor apparently cannot be checkpointed. In addition, portability is limited for a checkpointing system that must run on the Fluke microkernel.

System-level checkpointing

Many of the system-level checkpointing systems in existence today are focused on checkpointing parallel applications running on distributed operating systems or clusters of computers. MOSIX [17] is one such system. MOSIX is a distributed operating system with checkpointing and process migration support designed for load balancing. MOSIX was designed for scalable clusters of PC's. While MOSIX does well in

achieving its goal of resource sharing, it does not provide much support for the System Administrator wanting to checkpoint a user's process before killing it. Simply put, MOSIX does not provide support for storing a checkpoint image in a file.

A similar system is CKPM [18]. CKPM was designed for checkpointing parallel applications running in a network-based computing environment. CKPM was also designed on the system-level but requires additional libraries for wrapping the Parallel Virtual Machine (PVM) libraries. CKPM's checkpointing strategy uses pessimistic log-based protocols. While this system is efficient in achieving a global consistent state when checkpointing parallel applications, it does not provide the ability to store a checkpoint image in a file. Furthermore, checkpointing is restricted to the parallel applications of the PVM.

One of the more notable and recent attempts to perform system-level checkpointing, that also provides the functionality we're looking for, was with a tool called epckpt [19]. Epckpt was designed with a focus on being able to checkpoint parallel applications; specifically, those parallel applications resulting from a parent process calling *fork()*. Epckpt comes as a patch for the Linux kernel. Since epckpt is part of the kernel, epckpt has a number of advantages over the checkpointing systems we have discussed so far. Processes to be checkpointed with epckpt need not be recompiled or even relinked with any special libraries. Furthermore, epckpt can handle a broader range of applications, since it is not dependent on any special language or compiler. Epckpt, being part of the kernel, has direct access to a process's address space. Furthermore, epckpt can write the checkpoint image to a number of file descriptor abstractions. Epckpt

can write the checkpoint image to a file, pipe, or socket. This enables epckpt to migrate a process at the time of the checkpoint.

One of the main disadvantages of system-level checkpointing is that it usually results in a rather large checkpoint image. Some refer to this approach as being a *core dump*. A core dump suggests a checkpoint image with much unnecessary information. Epckpt has directly addressed the issue of checkpoint size. Epckpt allows the user to omit shared libraries and binary files from the checkpoint image. In cases where the user knows that these files will still be available when the process is restarted, the size of the checkpoint can be greatly reduced. In some cases, the size of the checkpoint image was reduced by more than half when the shared libraries and binary files were omitted.

Epckpt does still suffer from a few disadvantages. Like a number of the checkpointing systems we have previously discussed, epckpt requires that the user know a process may need to be checkpointed, before starting that process. Epckpt consists of a new system call, *collect_data()* that notifies the kernel to start recording information about a process. This information consists of file names and libraries. Epckpt provides a tool called spawn that can invoke this system call when starting a process, but the limitation still exists. In addition, epckpt is designed as a patch for the Linux kernel. In order for one to use this system, they must recompile their entire kernel.

CRAK. A number of ideas used in the design of epckpt have been extended on in the work on CRAK [20]. CRAK is a Linux checkpointing system that has been designed as a kernel module. As a kernel module, CRAK enjoys a number of the same benefits as epckpt does by functioning on the system-level. CRAK has access to a process's address space in addition to a number of kernel objects. Furthermore, CRAK requires no special

libraries, compilers or any modifications to the user code. In addition, CRAK requires no special logging by the kernel or by any user-level application. The main limitation of CRAK is that it requires an operating system to provide module support. Without module support, CRAK cannot be loaded. However, once loaded, CRAK can checkpoint and restart a wide range of applications.

CRAK [20] provides two essential user-level tools: *ck* and *restart*. To perform a checkpoint with CRAK, one invokes *ck*. *Ck* accepts two command line arguments. The first command line argument is the process id of the process to be checkpointed. The second command line argument is the filename of the file where the checkpoint image should be stored. This user-level application then invokes the module functions that perform the necessary actions to create a checkpoint image. These module functions begin by stopping the execution of the specified process. Once the process is stopped, the following process information is copied into the checkpoint image file.

- Address space.
- Register set.
- Opened files.
- Pipes.
- Sockets.
- Current working directory.
- Signal handler.
- Termios information.

Just like with epckpt, the size of this checkpoint file can be greatly reduced by omitting the shared libraries and binary code. CRAK offers the same flags for omitting these items when possible. Once checkpointed, the checkpoint image files can be stored for a later restart or moved to another node to achieve process migration.

To restart a process, one simply invokes *restart*, and provides the filename of the file containing the checkpoint image on the command line. Restart works much like the

system call *execve()*. The address space data in the checkpoint image file is copied into the address space of the *restart* program. This in conjunction with restoring the other items from the list above essentially restores the process.

Undoubtedly, the work on CRAK has laid the framework for an excellent checkpointing system. To the knowledge of this author, no other checkpointing systems have attempted to handle items such as networked sockets the way CRAK has. However, CRAK has left a number of areas open for continued work. For example, support for items such as opened files, pipes and sockets exist at the user-level. We believe support for these items should exist at the system-level. Furthermore, there are a number of issues not supported by CRAK. The following is a list of items not supported by CRAK.

- Restoring of PID.
- A PID reservation system for checkpointed processes.
- CRAK only supports saving opened files' pathnames. No support for storing an opened file's contents.
- No support for handling opened files that have been deleted or modified.
- Does not restore file pointer when restarting a process.
- Only supports TCP Sockets. No support for UDP sockets.
- Does not support loopback address.
- Cannot restore an established TCP connection that is the result of a call to accept(). These established TCP connections are multiplexed on the same port as a listening socket. A listening socket is created by a call to listen().
- Always restarts a process in the same terminal window as the *restart* program. No support for restarting a process in another terminal window.

As discussed in later chapters, the author has expanded the work on CRAK to include support for the items in the above list. In addition, the author has moved support for items such as opened file, pipes, and sockets to the system-level.

Buffer Overflow Attacks

Background

A buffer overflow takes place when a larger amount of data is copied into a buffer than that buffer was designed to hold. Functions such as strcpy, strcat, sprintf, and gets do not perform bounds checking and thus allow programmers to write code that overflows. A buffer overflow attack is usually the result of a malicious individual purposely overflowing a buffer with the goal of altering a program's intended behavior.

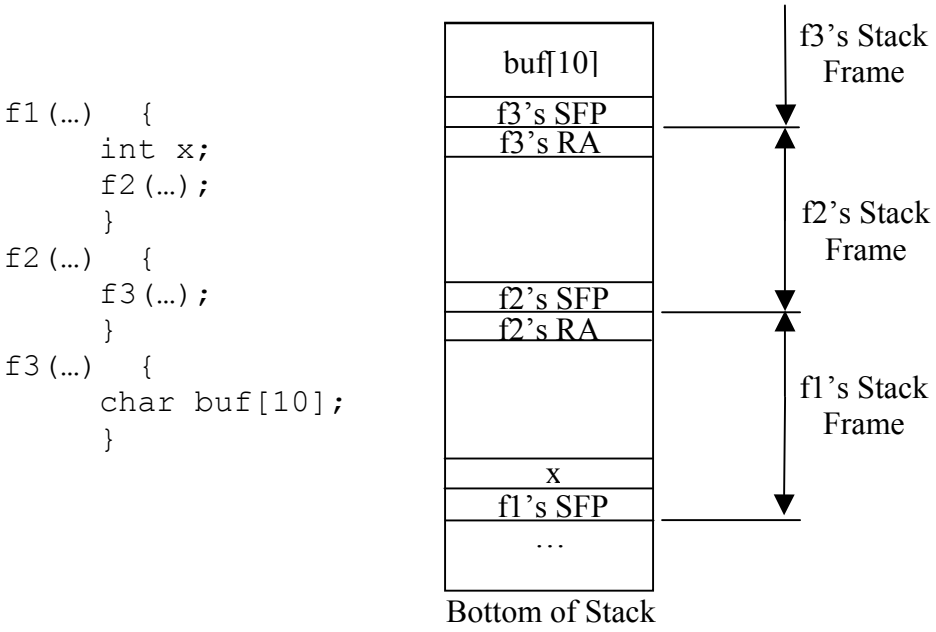

Figure 2-1. Typical runtime program layout.

Recall, that a function's return address resides on the stack just below a function's local variables. The only item between a function's return address and local variables is the saved frame pointer. This concept is shown in Figure 2-1. In the case where an array is declared as a function's local variable, such as *buf* in Figure 2-1, space for the array is allocated on the stack. If this array, or buffer, is overflowed with more data than was allocated for it, this excess data overwrites other items on the stack. For example, a skilled attacker may overflow *buf*, and thus overwrite *f3's* SFP and RA. If this address is

overwritten properly, when *f3* returns control flow of the program is sent to the location of the attacker's choosing. Commonly, the attacker sends control flow to a location on the stack where the attacker has injected his/her own malicious code. Usually the malicious code is injected inside the same array or buffer that is being overflowed. The result of this buffer overflow is that the attacker is able to execute his/her malicious code with the privileges of the original program. If the compromised program has root privileges, then the code executed by the attacker also has root privileges. One of the most common goals of an attacker launching a buffer overflow attack is to spawn a root shell. The code to spawn a shell is short and can be injected into a buffer rather easily. The impact of a malicious user gaining access to a root shell is beyond the scope of this paper, but clearly undesirable for any system administrator.

A buffer overflow attack that takes place on the stack is often referred to as a stack-smashing attack. Other types of buffer overflow attacks can take place in the heap, bss, or data segments. A buffer overflow that takes place in the heap is also referred to as a heap smashing attack. However, the stack-smashing attack is the most popular since it allows the attacker to inject code and alter control flow in one step. Other forms of stack-smashing and buffer overflow attacks involve redirecting control flow to other preexisting functions or even library functions.

Related Work

Stackguard

One of the most notable approaches to detecting and preventing buffer overflow attacks is referred to as StackGuard. Cowan et al. [21] created a compiler technique that involves placing a *canary* word on the stack next to the return address. This canary word acts as a border between a function's return address and local variables. It is very

difficult for an attacker to overwrite a return address without overwriting the canary word. When a function is returning, Stackguard checks to make sure the canary word has not been modified. If the canary word is unmodified then that implies the return address is also unmodified. One of the advantages of Stackguard is that it does not require any changes to the program source code or existing operating system. Stackguard does suffer from a performance penalty. However, that performance penalty is minor. The only downfall of Stackguard is that programs are only protected if they have been recompiled with a specially enhanced compiler.

Libsafe and Libverify

Baratloo et al. [22] proposed two new methods referred to as Libsafe and Libverify. Both methods were designed as dynamically loadable libraries and were implemented on Linux. Libsafe uses saved frame pointers on the stack to act as upper bounds when writing to a buffer. Libsafe intercepts library calls to functions such as strcpy() or scanf() that are known to be exploitable. It then executes its own version of these functions that provides the user the same functionality but also supplies the bounds checking based on the upper bounds set by the saved frame pointers. Libverify uses a similar approach to Stackguard in that a return address is verified before a function is allowed to return. Libverify does this by copying each function into the heap and overwriting the original beginning and end of each function with a call to a wrapper function. The wrapper function called at the beginning of a function stores the return address, allowing the wrapper function called at the end of the function to verify the return address. One downfall of this method is that the amount of space in memory required for each function is double that of what the process would require if not using Libverify.

VtPath

One approach proposed by Feng et al. [23] is VtPath. VtPath is designed to detect anomalous behavior but would also work well in detecting buffer overflow attacks. VtPath is unique in that it uses information from a program's call stack to perform anomaly detection. VtPath intercepts system calls. At each system call it takes a snapshot of the return addresses on the stack. The sequence of return addresses found between two system calls creates what is referred to as a virtual path. During training, VtPath can learn the normal virtual paths that a program executes. When online, VtPath detects any virtual paths that were not experienced in training. When such a path occurs, it is likely that an anomaly has occurred.

RAD

Another proposed approach to defend against buffer overflow attacks was introduced by Prasad et al. [24]. This approach involves rewriting binary executables to include a return address defense (RAD) mechanism. This approach is rather complex since it requires accurate disassembly in order to distinguish function boundaries. Once function boundaries are located they can be rewritten to include the RAD code. Upon entering a function, the RAD code stores a second copy of the return address that is later used to verify the return address on the stack when a function returns. Unfortunately, this approach is limited due to the challenges faced in disassembly. As Prasad points out, distinguishing between code and data in the code region can be an undecidable problem. Furthermore, we suspect this approach could lead to significant overhead during runtime when the ratio of lines of code to the number of functions decreases.

Static analysis

Other approaches aimed at the broader issue of anomaly detection include the call graph and abstract stack models proposed by Wagner et al. [25]. These methods use static analysis of program source code to model the program's execution as a nondeterministic finite automaton (NDFA) or nondeterministic pushdown automaton (NDPDA). These methods monitor a program's execution while using these models to determine if the sequences of system calls generated by a program are consistent with the program's source code. The downfall of these approaches is that they require access to a program's source code. When dealing with legacy applications or commercial software access to program source is often unavailable.

Computer Forensics

Stephenson defines *computer forensics* as the field of extracting hidden or deleted information from the disks of a computer [26]. Carrier [27] refers to computer forensics as the acquisition of hard disks and analysis of file systems. Simply put, computer forensics is the art of extracting digital evidence from a computer system usually associated with a crime. Not relevant to this discussion, computer forensics does at times include rescuing data from a damaged or corrupted computer system. Commonly, a computer forensic investigation takes place on the computer system that has either suffered an attack from another computer, or on the apprehended computer of a suspected criminal. In the case of the computer attack, the forensic investigator is usually attempting to find evidence that can answer questions such as, where did the attack originate, what vulnerability made the attack possible, and what files were compromised as a result of the attack. In the case of the suspected criminal's apprehended computer, the forensic investigator is usually looking for evidence of the suspected criminal's recent

behavior, motives, or planning of future crimes. In either case, the forensic investigator has a number of tactics for collecting such evidence. The investigation may involve anything from searching the file system for incriminating text files to analyzing log files for evidence of the attack. Typically, a computer forensic investigation involves using special forensic tools to analyze items such as slack space, unallocated space, or swap files. Slack space is the leftover space in a block or cluster allocated to a file but not used by the file. Unallocated space is space that is currently not used by any file. Both of these items may contain bytes from old files that were deleted but have yet to be fully overwritten. This allows a digital forensic investigator using forensic tools to extract this data. Swap files can be thought of as scratch paper for an application or the operating system. These files may have traces of data that allow the digital forensic investigator to piece together what actions have previously taken place on the given computer. Another example of data often used in a forensic investigation that does not require special tools to extract would be log files. During a forensic investigation, log files on the victimized or suspected computer are of the utmost importance. A survey of the literature on computer forensics reveals the direct correlation of logging and a successful forensic investigation [26,28,29]. Stephenson refers to the lack of logs as the single biggest barrier to a successful investigation of an intrusion. Upon completion of a forensic investigation all of the extracted evidence is preserved and stored in a secure facility. A *chain-of-custody* is maintained to assure that no one tampers with the collected evidence. A chain-of-custody is simply a system of recording who is responsible for the evidence at any point in time from the moment it was collected till the moment it is used in a courtroom.

Slack space, unallocated space, swap files, log files, and most other items analyzed by the forensic investigator shared an important similarity. Each of these items exists as nonvolatile data. Nonvolatile data is that which has been saved to disk or resides on some form of stable storage. The opposite of nonvolatile is obviously volatile. Volatile data is that which resides in main memory such as a process's address space. Once a computer is unplugged from its power source, all volatile data is lost, but nonvolatile data remains intact. Due to this inherent nature of digital data, computer forensics is largely restricted to the analysis of nonvolatile data. We believe one of the major keys to improving and enhancing computer forensics is to increase the amount of relevant nonvolatile data available to the forensic investigator. Later in this paper we discuss the idea of using checkpointing technology to create additional nonvolatile data from one of the most common forms of volatile data, namely, processes. This would add checkpoint image files to the collection of items the forensic investigator can analyze for evidence. As we know from previous sections of this chapter, a checkpoint image file contains a plethora of information.

CHAPTER 3
CHECKPOINTING

A Robust Checkpointing System

The ideal checkpointing system should be able to handle a wide range of issues. It should not only meet the needs of process migration in a distributed computing environment, but also provide checkpointing support for the system administrator who wishes to checkpoint a user's seemingly runaway process rather than kill it. Developing a checkpointing system that can support both these issues and the wide range of issues in between is not a simple task. To aid us in this task we have isolated the Three AP's to Checkpointing, namely, the ideal checkpointing system should support checkpointing for: Any Process on Any Platform at Any Point in time [30]. The Three AP's to Checkpointing serve as our guide in working towards the ideal checkpointing system.

More specific requirements of the ideal checkpointing system could be classified into three categories: transparency, flexibility, and portability. One of the most common forms of transparency sought after in a checkpointing system is transparency for the application programmer. The ideal checkpointing system would not require any modifications to the application programmer's code to support checkpointing. In addition, the application programmer would not be restricted to any special language or compiler. Furthermore, the application would not have to be recompiled or even relinked with any special checkpointing libraries. The ideal checkpointing system would also be transparent to the operating system code. In other words, it would not require any modifications to the operating system code. In addition, this system would not require

any logging of information. Furthermore, no system call wrappers would be required. Flexibility would be achieved by having the system support all possible aspects of a process including PID, opened files, pipes, and sockets. Further flexibility could be achieved by allowing the user to include or exclude items from the checkpoint image. This would allow the user to reduce the checkpoint size when necessary. Examples of items that should be able to be included or excluded from a checkpoint are shared libraries, binary files, and the contents of opened files. The system should also allow the user to restart a process in whatever terminal or pseudo-terminal the user wants. Portability is achieved when the checkpointing and/or restarting can take place on a number of different types of systems.

To date, no checkpointing system has been able to satisfy all the requirements mentioned in the previous paragraph. This would obviously imply that no system has ever supported all three AP's. While developing a checkpointing system that supports all three AP's and satisfies all the previously mentioned requirements is quite difficult, it is in fact our long term goal. However, our more immediate goal, and for the purpose of this paper we are focusing on two of the three AP's; Any Process at Any Point in time. The checkpointing system developed here operates on the system-level and is designed as a kernel module. This system satisfies these two AP's and meets the vast majority of the requirements listed in the previous paragraph. Our system expands the work of CRAK [20] to work on more recent and stable versions of the kernel and include additional functionality. Our system is called UCLiK, (Unconstrained Checkpointing in the Linux Kernel).

We believe that such a system should operate at the system-level for a number of different reasons. One of the most compelling reasons is that it gives us access to all the kernel and process state information. Without this information, other systems have had to rely on system call wrappers to perform logging of process information. In addition, checkpointing at the system-level aids our quest for transparency. It completely alleviates the application programmer from any responsibility. In addition, no special compiler or checkpointing libraries are needed. Furthermore, by having direct access to kernel functions and data structures we are more efficient than if we were executing at the user-level, incurring additional overhead from user code and library routines.

Our only disadvantage is portability. Since our system functions on the system-level, it is not very portable to other types of operating systems. Hence, the second AP, Any Platform, is not satisfied at this time. However, it is important to point out that the major benefit of the second AP, Any Platform, would be cross-platform migration. At this time, cross-platform migration is not one of our goals. Furthermore, this type of migration would be extremely difficult considering how differently checkpoints would be created on different platforms. It is very likely that any single system able to support cross-platform migration would suffer greatly in areas such as transparency and flexibility, and development of a completely portable system may ultimately be impossible. Table 3-1 shows how the different checkpointing systems addressed in the previous chapter measure up to the AP's of Checkpointing. We can see in this table that no system is successful in addressing the AP, Any Platform. We can also see where our system, UCLiK, is more successful at satisfying the other two AP's, Any Process and Any Point in time than any other existing checkpointing system.

Table 3-1. AP table for the different checkpointing systems.

	Any Process	*Any Point in time*	*Any Platform*
CHK-LIB	No. Processes must be linked with the run-time library.	No. Checkpoints are only made at the points specified by the programmer.	No.
Condor	No. Processes must be linked with the run-time library and run within the Condor system.	Yes.	No.
Libckpt	No. Processes must be recompiled with the checkpointing libaray and main() must be renamed to ckpt_target().	No. Checkpoints are created a prespecified time intervals.	No.
ZPL	No. Processes must be written with the ZPL language and compiled with the ZPL compiler.	No. Checkpoints can only be created during certain ranges of code.	No.
MOSIX	No. Processes must be part of the MOSIX cluster system.	Yes and No. A globablly consistent state must achieved. Not targeting a single process.	No.
CKPM	No. Requires libraries for wrapping the Parallel Virtual Machine (PVM) libraries.	Yes and No. A globablly consistent state must achieved. Not targeting a single process.	No.
Epckpt	No. The new system call, collect_data() must be invoked to collect data about a process to be checkpointed.	Yes.	No.
CRAK	Almost. CRAK has the freedom to checkpoint any process but does not provide support for items such as the PID, opened file's contents, file pointers, opened files that have been deleted or modified, UDP sockets, and loopback addresses.	Yes.	No.
UCLiK	Yes. UCLiK has the freedom to checkpoint any process and provides support for those items CRAK does not.	Yes.	No.

In conclusion, we believe checkpointing at the system-level brings us closer to achieving the ideal checkpointing system than any other approach. Checkpointing at the system-level provides transparency that is unmatched by user-level checkpointing systems. It is also important to note that by developing the system as a kernel module additional transparency is achieved. Since a module can be inserted and removed from the kernel code, modifications to the running kernel's code is unnecessary. Furthermore, we can provide more flexibility with our checkpointing system than has been supported by other systems. Additional issues of portability and potential standardization of checkpoint image files is addressed in a later chapter.

UCLiK Overview

To best understand how UCLiK works, it is best to first understand the files involved with UCLiK. The UCLiK system is primarily composed of the following four files.

- ukill.C
- ucliklib.c
- uclik.c
- uclik.h

The file *ukill.C* is the source code for our user-space tool. This tool is what a user would invoke to perform a checkpoint, or to restart a checkpointed process. The file *ucliklib.c* contains helper functions that assist the user-space tool in communicating with the functions in the kernel module. The kernel module source code is located in the third file, *uclik.c*. The fourth file, *uclik.h*, is a header file shared by all three of the previously mentioned files.

The user-space tool communicates with the module through a device file. The name of this device file is */dev/ckpt*. This device file is created when the module is

loaded into the kernel. The module registers itself as the owner of this device file. For simplicity, we refer to the user-space tool as being able to call functions in the kernel module. However, for preciseness we explain here that the user-space tool actually calls functions in *ucliklib.c*. These functions then make *ioctl()* calls to the device file, */dev/ckpt*, which is owned by the kernel module. Thus the kernel module receives these *ioctl()* calls and directs them to the corresponding kernel module functions. Figure 3-1 illustrates this concept more clearly.

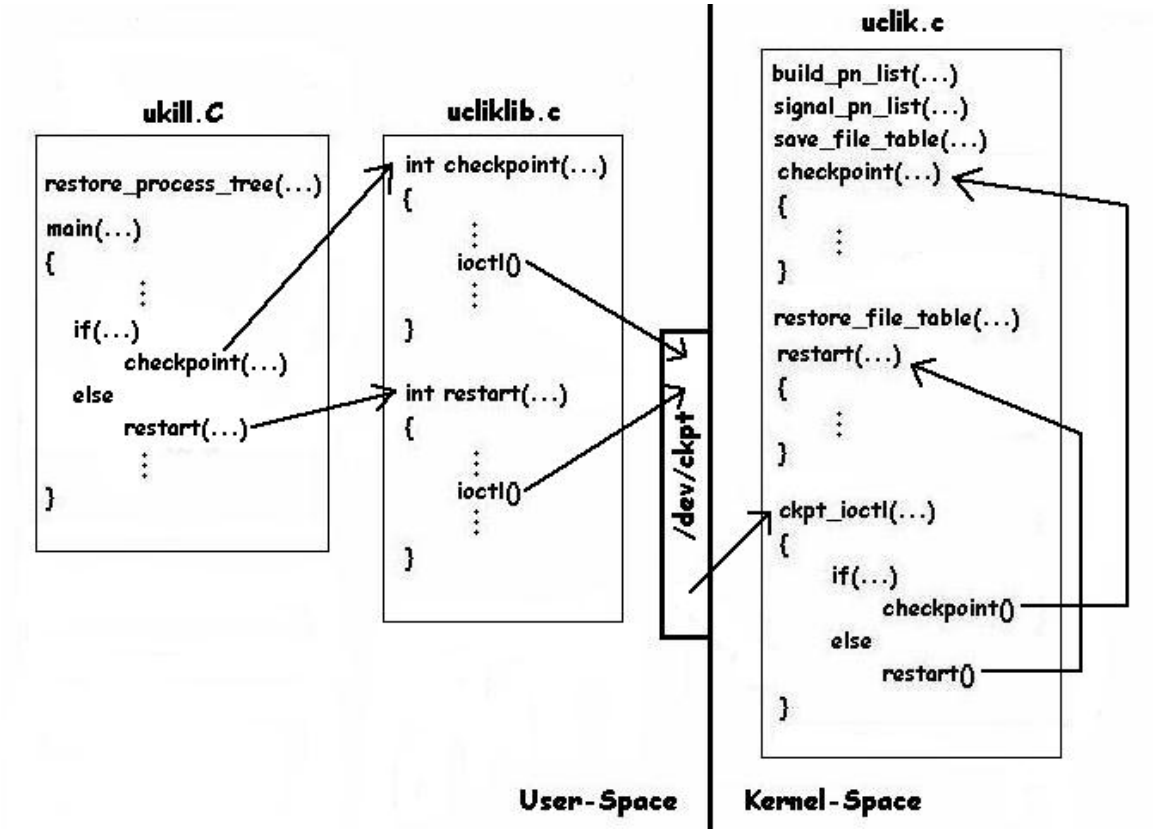

Figure 3-1. Performing a checkpoint with UCLiK.

A checkpoint is created when a user invokes the user-space tool *ukill*. The *ukill* tool receives a minimum of one command line argument. This command line argument is the PID of the process to be checkpointed. By default, the process's checkpoint will be saved in a file named with the process's PID. The user can optionally include a filename

as an additional command line argument to specify where the checkpoint image should be stored. When checkpointing, the main purpose of *ukill* is to open the file where the checkpoint should be stored, and pass it's file descriptor to the *checkpoint()* function in the kernel module. Ukill also passes the PID number and any additional flags sent on the command line to the *checkpoint()* function in the kernel module. The *checkpoint()* function will use the PID number to locate the given process's process descriptor (represented with a *task_struct* structure). The *checkpoint()* function will immediately send the SIGSTOP signal to the process being checkpointed. If we are checkpointing a family of processes, the SIGSTOP signal will be sent to each process in that family. Once the process is stopped, the real checkpointing begins. UCLiK begins by storing crucial fields of the process descriptor such as pid, uid, gid, and so forth. Following this, the process address space is stored. This includes fields from the memory descriptor (represented with an *mm_struct* structure) such as initial and final addresses of the executable code, initialized data, heap, command-line arguments, and environment variables. The initial address of the stack is also saved. Next, UCLiK loops through each of the memory regions (represented with a *vm_area_struct* structure). For each memory region, the linear address boundaries, access permissions and flags of that region are saved. The contents of each memory region are also saved. Following this, UCLiK iterates through each opened file descriptor saving necessary information for each file abstraction. UCLiK provides support for opened files, pipes, and sockets. Lastly, UCLiK stores information about items such as the current working directory and the signal handler. Once all of this information is written to a file, UCLiK can optionally kill the process and then exit.

A process is restarted by invoking the user-space tool *ukill* and activating the undo switch. When the undo switch is activated, the command line argument following the undo switch should be the name of the file containing the image of the checkpointed process. Very often, this will simply be the process's original PID (i.e., ukill –undo PID). *Ukill* will read this file to find out process family information. If it is a single process, then *ukill* will call the *restart()* function of the kernel module. If it is a family of processes, then *ukill* will fork the appropriate number of processes, in the appropriate order, and allow each new process to call the *restart()* function separately. The *restart()* function of the kernel module will subsequently read in all the process information from the checkpoint image file, copying the original process's information over the *ukill* process. Once the kernel module completes this task, it allows the process to run. A family of processes is handled slightly different and is addressed later.

UCLiK's Comprehensive Functionality

Opened Files

CRAK's handling of opened files was very straightforward. During a checkpoint, the following information is stored as part of the checkpoint file.

- File pathname.
- File descriptor number.
- File pointer.
- Access Flags.
- Access Mode.

During a restart, CRAK opens the file using the file pathname, access flags, and access mode. This open takes place at the user-level. If opening the file is successful, then the file is duped if necessary to assure it had the same file descriptor number as

before the checkpoint. This approach worked well but left a number of issues open for continued work.

One of our objectives is to provide transparent checkpointing at the system-level. During restart, CRAK would reopen files at the user-level. With UCLiK, during a restart, files are opened at the system-level. One major advantage of reopening a file at the system-level is that it allows us to also restore the file pointer. The following subsections address a number of other issues with opened files such as restoring the file pointer. All of these issues are also being dealt with at the system-level.

Restoring the file pointer

UCLiK, unlike its predecessor, has the ability to restore the file pointer. To understand the importance of a file object's file pointer, we must first understand how a read() system call made in user-space is handled by the kernel. As an example, suppose we have a file that is 14,361 bytes in length. Suppose we also have a process that is going to read this entire file line by line.

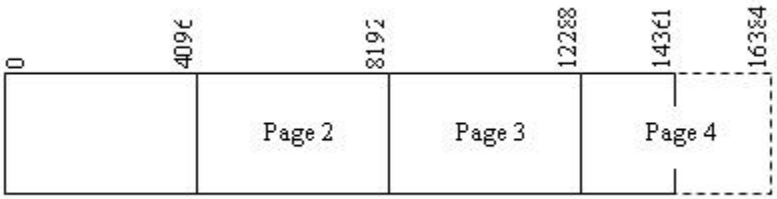

Figure 3-2. File size relative to page size.

We assume the user-space process has already opened the file. At the point when the user-space process first begins to read the file, the kernel loads the first page of the file into the process address space of the user-space process. The file object pertaining to the file being read has its file pointer (i.e., the *f_pos* field) assigned the value 4096. This seems strange since the file is being read line by line. But the buffering of line by line

reading is actually handled by *glibc*. The *glibc* portion of execution takes place in user-space, but is hidden from the programmer. The following image illustrates the relationship of the process, glibc, and the kernel.

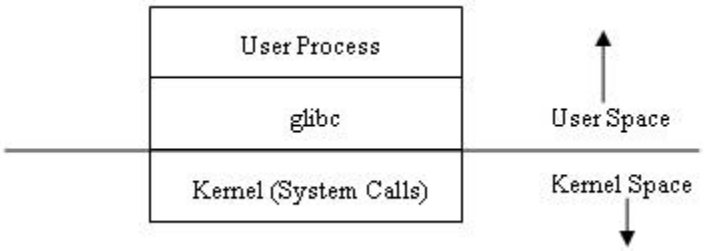

Figure 3-3. Glibc executes in user-space.

Inside the glibc layer there are three variables of great importance: _IO_read_base, _IO_read_ptr, _IO_read_end. When the page is loaded into memory, the variable _IO_read_base points to the first byte of the page. The variable _IO_read_end points to the first byte just after the last byte of the page. Meanwhile, while the file is being read line by line, variable _IO_read_ptr points to the next unread byte in the page. Essentially, _IO_read_ptr is our real file pointer.

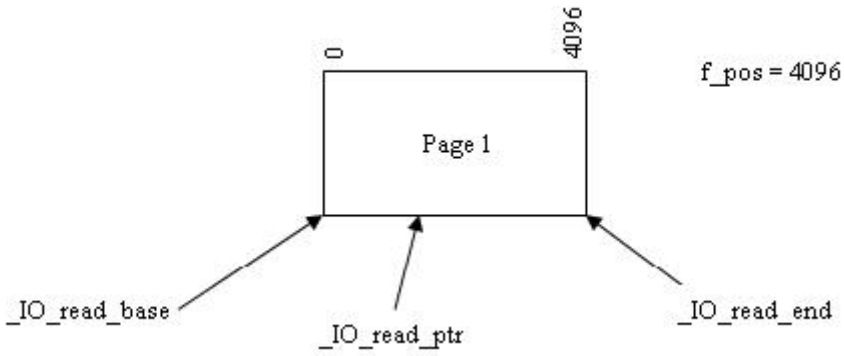

Figure 3-4. Glibc variables during a read.

_IO_read_ptr is modified with each incremental read. Once the value of *_IO_read_ptr* equals the value of *_IO_read_end*, the kernel removes this page from the

process's address space, and loads the next page (Page 2) into the process's address space. At that point, the *f_pos* value is assigned the value of 8,192. This process continues until the end of the file is reached.

For the purposes of process checkpointing and restarting it is essential that the value of *f_pos* be saved and restored when the process is restarted. Let us continue our above example of a process reading a file of size 14,361 bytes, only this time we checkpoint the process and restart it without restoring the *f_pos* field. We assume the process was checkpointed when 4,363 bytes of the file had been read. This would imply that the second page of the file had been loaded into memory. The following image should illustrate this our current state.

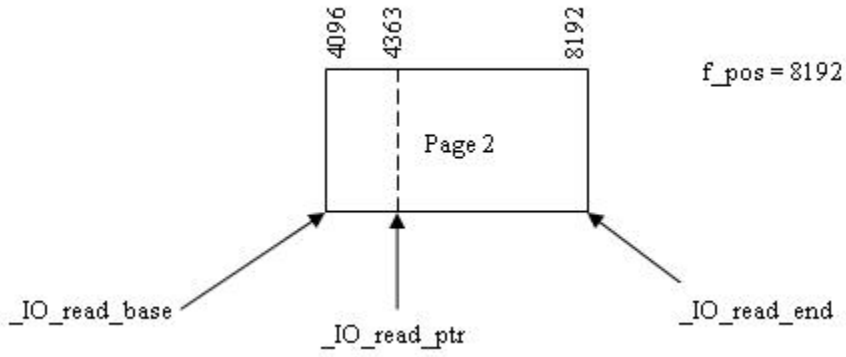

Figure 3-5. The second page of a file loaded into memory.

If the process is checkpointed at this moment, when the process is restarted, the *f_pos* value must be assigned a value of 8192. If it is not, then it is assigned a default value of 0. Meanwhile, our glibc values, *_IO_read_base*, *_IO_read_ptr*, *_IO_read_end* will be restored to their original values. Since the glibc values are part of the process address space, when the process address space is restored, the glibc values are also restored. When reading is continued, the *_IO_read_ptr* value causes the reading to pick

up where it left off. However, when the value of *_IO_read_ptr* equals that of *_IO_read_end*, the kernel then uses the *f_pos* value to determine which page to load into memory next. Since the value of *f_pos* is 0, the kernel loads the first page of the file into the process's address space. From here, the entire file is read.

To summarize, when a process is restarted, if the *f_pos* value is allowed to default to 0, then the rest of whatever page was already loaded into the process' address space is read. This is then followed by the entire file being read.

In testing, because the *f_pos* value was allowed to default to 0 the resulting number of bytes read after restarting the process was equal to the number of bytes of the file plus the number of unread bytes in the page that was loaded into memory at the time the process was checkpointed. For example, the file being read was 14,361 bytes in length. The process was checkpointed when 4,363 bytes had been read. This meant that the second page was the page currently loaded into the processes address space. This also meant that there were 3,829 bytes remaining in that second page that had never been read. When the process was restarted, the number of bytes read after the restarting was 18,190. This is the sum of 14,361 and 3,829.

Furthermore, it is important to note that the *f_pos* value must be restored in kernel space. One could attempt to restore the *f_pos* value from user-space through the use of functions like *lseek()*. However, if this is done from user-space the *_IO_read_ptr* value of the *glibc* layer is also modified. As we have seen above, the *f_pos* value is usually greater than the *_IO_read_ptr* value. The *f_pos* value points to the end of the page currently loaded in the process's address space. If the *_IO_read_ptr* value is restored from user-space, the only value available is that extracted from the *f_pos*. If

_IO_read_ptr is restored to the value of *f_pos*, we more than likely, skip some portion of the file being read.

File contents

If we consider the list of items recorded in a CRAK checkpoint we immediately see another important missing file attribute. The actual contents of an opened file were not included in the checkpoint image. In UCLiK we have added support for packaging the file contents of opened files in the checkpoint image. Since this can drastically affect the checkpoint size we leave this as an option for the user. When invoking u*kill*, the user can specify a flag, *-p*, that notifies the system to package the contents of opened files with the checkpoint image. Later, when invoking *ukill -undo*, the user can specify the same flag, *-p*, to unpack the files that have been packaged with the checkpoint image. Unpacking the files obviously just creates a copy of the files. If the original files are not available, the user can specify another flag that tells the system to use the new copies of these files.

Deleted and modified files

Another issue left open by CRAK was how to handle deleted and modified files. Having already added the functionality to package the contents of opened files with a checkpoint image, this issue is resolved rather easily. During a restart with UCLiK, if a file is found to be missing or modified since the time of the checkpoint, the restart alerts the user and cancels itself. At this point, the user can restart the process again using the flag that tells the system to force the restart. This restarts the process with the missing or modified file. Of course the user still has the option to unpack and use any packaged copies of files that were included in the checkpoint image file.

Restoring PID

CRAK makes no attempt to restore the PID. When CRAK restarts a process, the PID assigned to the *restart* process is then inherited by the restarted process. We have added the functionality of restoring a process's PID to UCLiK. We use the *find_task_by_pid()* function to determine if the PID is available. If it is available, then the restarted process is assigned the original PID. However, if the PID is not available, then the restarted process will be run with the new PID.

Pipes

CRAK's handling of pipes was rather similar to that of opened files. During a checkpoint, information such as inode number and whether a pipe was a reader or a writer was included in the checkpoint image. During a restart, a new pipe was created and duped if necessary to the correct file descriptor. The creation of this new pipe during restart took place at the user-level. We have moved this functionality to the system-level.

UCLiK recreates pipes at the system-level. However, creating pipes at the system-level incurs some additional complexity. Pipes are created with two ends, one for reading and one for writing. When a user-level application creates a pipe it is usually followed by a fork. The parent process can then close one end of the pipe, and the child process will close the other end of the pipe. This allows the two separate processes to share a single pipe. This pipe then acts as a one-way channel of communication. By the very nature of our restart mechanism this is hard to replicate. Our restart mechanism consists of the user-level *ukill* program whose address space is copied over with the address space of the checkpointed application. When we are restarting multiple processes, *ukill* forks and calls the restart function of our kernel module once for each process being restarted. Imagine if there are two processes being restarted, and there

exists a pipe between them. When the first process invokes the restart function in the kernel module, the pipe is created. When the second process invokes the restart function of the kernel module, it needs the same pipe. If it creates a pipe, it is a new pipe. We handle this scenario by passing an additional parameter to the restart function of the kernel module. This parameter is called *family_count*. The *family_count* tells the module how many parallel processes are being restarted. The module then knows to maintain pointers to any created pipes. Then when other processes need to access those same pipes, identified by their original inode numbers, the module still has access to them. This has enabled us to restore pipes from kernel-space. Additional detail concerning pipes between processes in larger groups of parallel processes is addressed in the next subsection.

Parallel Processes

One of UCLiK's most beneficial features is its ability to checkpoint parallel processes. While some checkpointing systems do not support checkpointing parallel processes, those that do are often constrained by the types of parallel processes they support. Some such systems only checkpoint parallel processes consisting of a single parent and its immediate children. UCLiK is not constrained by the type or number of parallel processes it can checkpoint. Once again, since UCLiK operates on the system-level it has access to process descriptor fields such as *p_cptr* and *p_osptr*. The *p_cptr* field of the process descriptor is a pointer to the process descriptor of the process's youngest child. A process's youngest child is the most recently spawned child. The *p_osptr* field points to the process descriptor of a process's older sibling. A process's older sibling is considered the process spawned by the same parent just before the spawning of the given process (Figure 3-6).

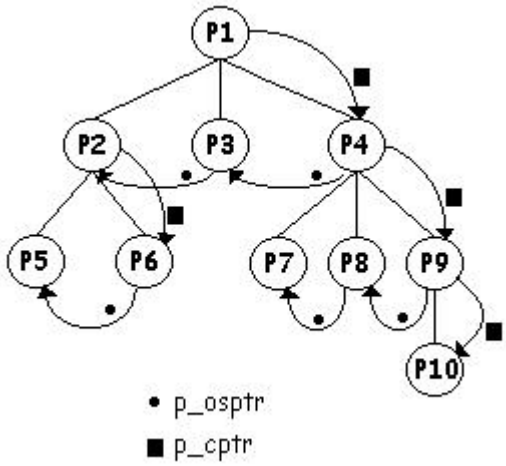

Figure 3-6. Process descriptor fields *p_osptr* and *p_cptr*.

UCLiK uses these fields to navigate through a tree of processes while creating a linear list of the processes. This linear list is stored as part of the process family's checkpoint. Upon restart, this linear list of processes is recursively scanned through to fork a process for each original process in the original process tree.

The first process to be entered into the linear process list is always the highest parent process. Using the process tree from Figure 3-6, P1 would be the first process entered into the list. At this point, P1 becomes our main list item. UCLiK would then follow P1's *p_cptr* field to find P4's process descriptor. Using the *p_osptr* fields, P4, P3, and P2 would subsequently be added to the list in that order. Now that P1's immediate children have been added to the list, the item in the list following P1 would now become the main list item. This would be P4. UCLiK would then follow the same procedure for adding P4's children to the list. After P4 is the main list item, the next item in the list, P3, then becomes the main list item (Figure 3-7).

The dotted lines in Figure 3-7 surround the processes that have recently been added to the list. The arrow indicates which process in the list is currently the main list item.

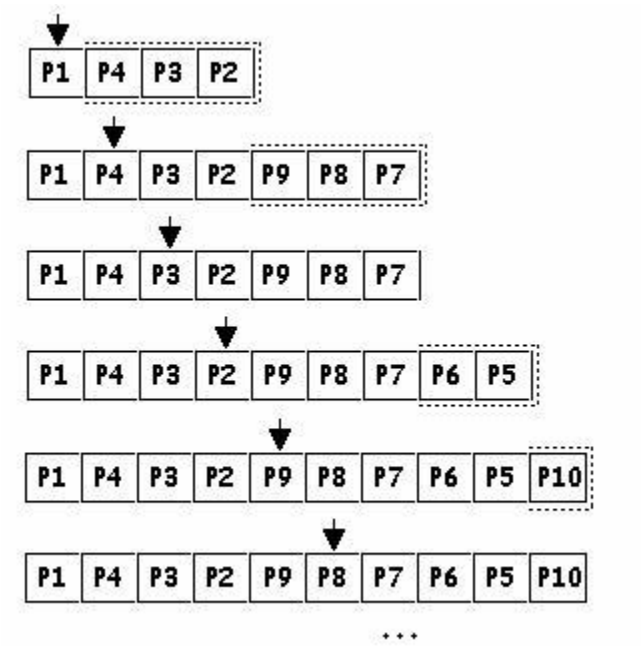

Figure 3-7. Building a linear list of processes.

We see in Figure 3-7 that when the arrow points to P3, that no processes are being added to the list. However, when the arrow points to P2, the processes P5 and P6 are then added to the list. This procedure continues until the arrow reaches the end of the list.

Once UCLiK has created a linear process list, it is easy for UCLiK to signal each process in the list to stop execution. This is done with the SIGSTOP signal. With each process in the group of parallel processes stopped, UCLiK can incrementally checkpoint each process to a file. Once all the processes are checkpointed, the execution of this family of processes can continue or be stopped.

Upon restarting a group of parallel processes, our user-space *ukill* tool is responsible for forking a process for each process in the original process tree. Recall, that all of these processes cannot be forked from a single process. We must have each process fork the same number of processes as its corresponding original process had forked. To facilitate this procedure we maintain two additional values for each process in

the list. These two values are referred to as *children* and *childstart*. The *children* value is simply the number of children a given process has spawned. The *childstart* value tells us what position in the list is a given process's oldest child. Table 3-1 contains the values for the process tree in Figure 3-6.

Position In List	0	1	2	3	4	5	6	7	8	9
Process	P1	P4	P3	P2	P9	P8	P7	P6	P5	P10
Children	3	3	0	2	1	0	0	0	0	0
Childstart	3	6	0	8	9	0	0	0	0	0

Table 3-2. Example values for children and childstart.

At first glance, the *childstart* values may not be apparently obvious. However, if we start at the beginning of the list with P1, and move through the list summing the *children* values, we quickly see where the *childstart* values come from. For example, adding P1's *children* value of 3 with P4's *children* value of 3, and we get P4's *childstart* value of 6. Since, P3 has zero children, its *childstart* value is also zero. However, adding P4's *childstart* value of 6 (which is the current sum of *children* values), to P2's *children* value of 2, and we get P2's *childstart* value of 8. We continue this process to fill in the rest of the table.

Once these values are determined, UCLiK uses this list to fork the appropriate number of children and thus recreating the original process tree. This procedure is done with a combination of iteration and recursion. A call to our *restore_process_tree()* function starts with P1 at the beginning of the list. This function uses P1's *children* and *childstart* values to iterate through P1's children, while forking a process for each one. This iteration moves in a right-to-left order relative to the process list shown in the top of Figure 3-8. For P1, the function forks a process for P2, then P3, and then P4. Each new

forked process then recursively calls the *restore_process_tree()* function on itself. This causes each forked process to have its own children iterated through the same way as P1. Subsequently, when *restore_process_tree()* is called for process P2, it will fork a process for P5 and P6. When this function is called for process P4, it will fork a process for P7, P8, and P9. Figure 3-8 illustrates this procedure. The arrows in Figure 3-8 indicate what process the *restore_process_tree()* function was called for. The dotted lines are used to represent what processes are being forked.

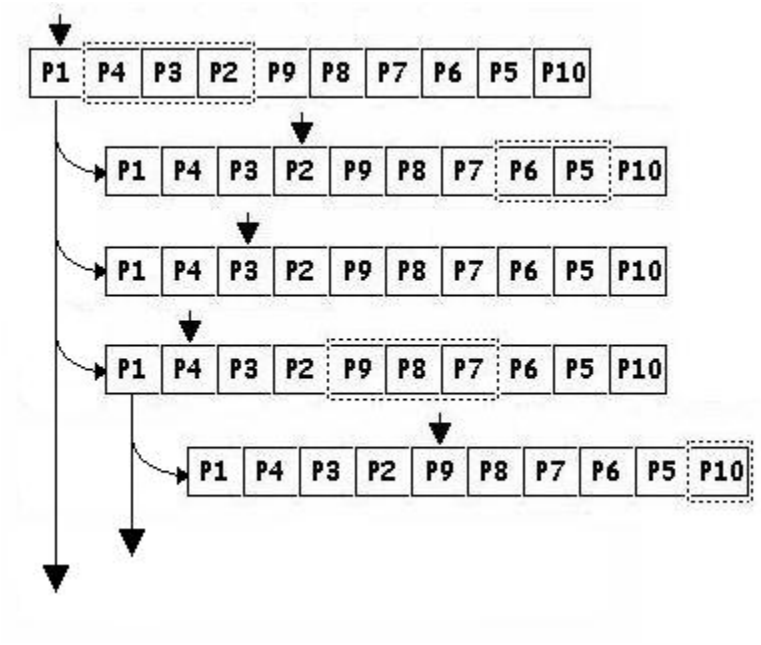

Figure 3-8. Building a process tree from a process list.

While the function *restore_process_tree()* is rebuilding our original process tree it is also invoking functions in the kernel module to restore each of the original processes. It makes a separate call to the kernel module for each process that must be restored. During this, UCLiK must keep track of how many processes are in a group of processes. For all the processes except the last one, UCLiK sends them the SIGSTOP signal after restoring their process address space and kernel state. When UCLiK restores the last

process in a group, it then sends all the rest of the processes the SIGCONT signal. From here, the group of processes can then continue their execution.

Restoring PIDs of parallel processes

During a checkpoint the original PID of each process is stored as part of the checkpoint. During a restart, if a process's original PID value is not in use then UCLiK has the ability to restore a process's original PID. When restarting a group of parallel processes, at the point in which UCLiK is restoring the first process of this group, UCLiK must also check to see if the entire group's original PID values are in use. If none of the original PID values are in use, then UCLiK can restore these PID values for the restarted group of parallel processes.

To check the availability of the group's original PID values, UCLiK makes use of the kernel function *find_task_by_pid()*. This function is called for each of the original PID values. If a particular PID value is in use, this function will return a pointer to the process descriptor of the corresponding process. If a particular PID value is not in use, this function will return NULL.

Restoring pipes between parallel processes

When checkpointing with UCLiK, we save pertinent information about pipes between parallel processes. Upon restart, UCLiK restores these pipes from kernel-space. Recall, that for a group of parallel processes, the *ukill* tool makes an individual call to the kernel module for each process being restarted. Considering that two different processes often share a single pipe, a pipe created for one process, must be accessible during subsequent calls to the kernel module. To handle this situation, we created a pipe cache. When a given process needs one end of a pipe, the pipe cache is always scanned first. If the pipe is not in the cache, then we create the pipe and add it to the cache.

UCLiK makes use of the *do_pipe()* system call to create pipes in kernel-space. The *do_pipe()* function receives as input a pointer to an array of two integers. When *do_pipe()* returns, these two integer values will correspond to the file descriptors of the file objects that represent the two ends of the pipe. These two file objects will have been installed in their corresponding file descriptor positions for the current process. A pointer to the inode shared by these two file objects and a pointer to each of these file objects are stored as an item in UCLiK's pipe cache. Items in the pipe cache are identified by the original pipe inode number. We refer to the original pipe inode as the inode of the pipe that existed before checkpointing the group of processes. In the case of two processes that share a single pipe, the first process to be restarted will cause UCLiK to create a new pipe. After creating this pipe, UCLiK will install the appropriate end of the pipe to the first process, and place a corresponding item in the pipe cache. When restarting the second process, the pipe cache identifies an item by the original pipe's inode number, and installs the pipe from the cache item.

A final note on our pipe cache is that entries in the cache should not carry over from one group of parallel processes to another group of parallel processes. To prevent this from happening, any call to the kernel module consists of an additional value referred to as *family_count*. The *family_count* value represents the number of processes in a group of processes. This allows UCLiK to determine when the first and last processes of a given group of processes are being restarted. When UCLiK determines that it is restarting the last process in a group of processes, it can then clear the pipe cache.

Restoring pipe buffers between parallel processes

UCLiK also has the ability to restore the pipe buffers between parallel processes. UCLiK makes use of the kernel's preprocessor macros PIPE_START, PIPE_BASE, and

PIPE_LEN. PIPE_START points to the read position in the pipe's kernel buffer. PIPE_BASE points to the address of the base of the kernel buffer. PIPE_LEN represents the number of bytes that have been written into the kernel buffer, but have not yet been read. During a checkpoint, UCLiK utilizes these macros to store a copy of the buffer along with the rest of the checkpoint. Later on during a restart, these same macros are used again to refill the buffer with the same contents it had before the checkpoint.

Sockets

CRAK stands out from most checkpointing systems by its ability to checkpoint and even migrate some sockets. However, much like opened files and pipes, when restarting a process consisting of sockets, these sockets were being created and even bound at the user-level. We have moved support for sockets from the user-level to the system-level.

One issue left open by CRAK was that of loopback addresses. CRAK did not support loopback addresses. UCLiK supports loopback addresses.

TCP sockets use a client/server relationship. A typical TCP socket connection is established by the following procedure. On the server side, a socket is created and then bound to a local address and port number. This is done with the use of the *socket()* and *bind()* C Library functions. The server can then begin listening using the *listen()* function on the port to which it is bound. On the client side, a socket is created and optionally bound. When the client invokes the *connect()* function, the corresponding server will accept this connection request with a call to the *accept()* function. If the client does not call *bind()* before *connect()*, then the client's socket is automatically bound to a random port. Once this procedure is complete an established socket exists between the client and the server. The interesting aspect to this procedure is that the server now has two sockets. One socket, that was created by the server and was set to listen on the bound port. A

second socket, which has an established connection with the client. When the server calls the *accept()* function, a second socket is created. This socket is multiplexed on the same port as the listening socket. Usually, different sockets at the same IP address must have unique port numbers, but in this case the port number is shared.

This creates an issue when checkpointing and restarting processes that contain sockets. Inside the Linux kernel a socket can be in any one of twelve different states. At this point, we only need to be concerned with three of these states, TCP_ESTABLISHED, TCP_CLOSE, and TCP_LISTEN. The other states are transitional states. A socket only exists in a transitional state during kernel mode execution that results from a system call. Before and after any of the system calls mentioned in the previous paragraph a socket will always be in one of the three previously mentioned states. Sockets that are part of an established connection will obviously be in the TCP_ESTABLISHED state. When restoring a socket in this state, the naive approach would be to bind the socket back to the port to which it was previously bound to. However, this does not work since the listening socket has already been bound to that port. CRAK handles this issue by allowing the established socket to be bound to another port. This works, but we have now restored the process with a socket that is not exactly like the socket it had before the checkpoint. Furthermore, this results in the client of this socket connection, needing to be notified that the server port for this socket connection has changed. A functionality that we do want to support, but we believe should be reserved for times when it is absolutely necessary, like process migration. In UCLiK, we utilize the function *tcp_inherit_port()* which allows us to remultiplex the

established socket onto the same port on which the listening socket is listening. We believe this is a better method.

Terminal Selection

We have also developed a tool that allows the user to restart a checkpointed process in the terminal or pseudo-terminal of their choice. This tool makes use of the ioctl() command to run a command in a different terminal window. The ioctl request of TIOCSTI makes it possible to write text to a different terminal window. This would be very helpful for the system administrator who wishes to restart a user's process in the user's terminal window rather than his/her own window.

CHAPTER 4
DETECTING STACK-SMASHING ATTACKS

The ICAT statistics over the past few years have shown at least one out of every five CVE and CVE candidate vulnerabilities have been due to buffer overflows. This constitutes a significant portion of today's computer related security concerns. In this paper we introduce a novel method for detecting stack-smashing and buffer overflow attacks. Our runtime method extracts return addresses from the program call stack and uses these return addresses to extract their corresponding invoked addresses from memory. We demonstrate how these return and invoked addresses can be represented as a weighted directed graph and used in the detection of stack-smashing attacks. We introduce the concept of a Greedy Hamiltonian Call Path and show how the lack of such a path can be used to detect stack-smashing attacks.

Overview of Proposed Technique

We propose a new method of detecting stack-smashing attacks that deals with checking the integrity of the program call stack. The proposed method operates at the kernel-level. It intercepts system calls and checks the integrity of the program call stack before allowing such system calls to continue. To check the integrity of a program's call stack we extract the return address and invoked address of each function that has a frame on the stack. Using the list of return addresses and invoked addresses we can create a weighted directed graph. We have found that a properly constructed weighted directed graph of a legitimate process always has the unique characteristic of a Greedy Hamiltonian Call Path (GHCP). We refer to this as a call path since it corresponds to the

sequence of function calls that lead us from the entry point of a given program to the current system call. This call path is greedy because when searching for this path within our weighted directed graph, we always choose the minimum weight edge when leaving a vertex. Furthermore, this path is Hamiltonian because every vertex must be included *exactly once*. Most significantly, we have found that the lack of such a path can be used to indicate that there has been a stack-smashing or buffer overflow attack.

Constructing the Graph

The task of constructing a weighted directed graph from the program call stack involves five major steps. We demonstrate these five steps on an example program. The functions, their source code, starting and ending addresses in memory for the example program are shown in Table 4-1.

Table 4-1. Example program we use to demonstrate graph construction.

Function Name	Starting Address in Memory	Ending Address in Memory	Function's Code
f3()	0x08048400	0x0804842a	execve(…);
f2()	0x0804842c	0x08048439	f3();
f1()	0x0804843c	0x08048449	f2();
main()	0x0804844c	0x0804845f	f1(); return 0;

The five major steps include

Step 1: Collect return addresses. Using the existing frame pointer, trace through the program call stack to extract the return address from each stack frame.

Step 2: Collect invoked addresses. For each return address extracted from the stack, find the *call* instruction that immediately precedes it in memory. Extract the invoked address from that *call* instruction. At this point we can create a table of return address/invoked address pairs. For the program shown in Table 4-1, the return and invoked addresses in Table 4-2 would be extracted.

Table 4-2. Return address/invoked address pairs.

Return Address	Invoked Address
0x08048321	0x42017404
0x42017499	0x0804844c
0x08048457	0x0804843c
0x08048447	0x0804842c
0x08048437	0x08048400
0x08048425	0x420b4c34
0x420b4c6a	0xc78b1dc8

In Table 4-2 it is easy to see how the addresses starting with 0x0804…. correlate to the addresses in Table 4-1. The addresses starting with 0x420…. are the addresses of C library functions used by our program. The last address, 0xc78b1dc8 is the kernel address of the system call function *execve*(). Addresses such as 0x420b4c34 and 0x420b4c6a correspond to the system call wrapper in our C library. The additional addresses at the beginning of the table (i.e. 0x08048321, 0x42017404 and 0x42017499) are the addresses corresponding to *_start* and *__libc_start_main*. The purpose of these functions is not pertinent to this paper.

Step 3: Divide addresses into islands. Once the values in Table 4-2 have been obtained we can begin construction of our weighted directed graph. Our final graph contains a node for each of the addresses in Table 4-2. However, before we can make each address into a node we must first categorize our addresses into what we refer to as islands. Our addresses are divided into islands based on their locations in memory. For example, addresses that begin with 0x0804… are part of a different island from addresses that begin with 0x420…. Addresses are further divided on whether they are a return address or an invoked address. In this example we have four islands. These islands are show in the Figure 4-1.

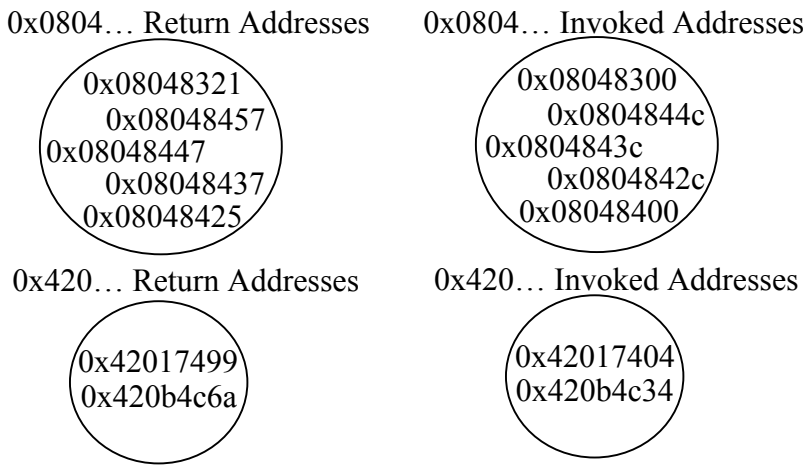

Figure 4-1. Example program's addresses divided into islands.

Note that the address 0xc78b1dc8 was not placed into an island. This is because this address represents the first instruction of our system call. It represents a unique node later on. We add this address' node when we begin adding edges. The address 0x08048300, the first instruction in the *_start* procedure, was added even though it is not in Table 4-2. This address is part of an ELF header and is loaded into memory, thus it can be extracted at runtime. Every program must have an entry point and therefore can be part of our graph.

Step 4: Adding edges. Recall the return address/invoked address pairs we have listed in Table 4-2. Each of these pairs are connected with a zero weight edge leading from the return address to the invoked address. At this time we can add a node for the address 0xc78b1dc8, and subsequently add a zero weight edge to it from it's corresponding return address. All of the edges added thus far are part of our final GHCP.

To complete this step, we attempt to give each invoked address node an edge to every return address node in the same memory region. These edges are weighted with the distance in memory between the two nodes. For example, the node with address 0x0804842c has a directed edge with weight 0x1b leading to the node with address

0x08048447. In addition, the node with address 0x0804842c also has a directed edge leading to 0x08048457 with a weight of 0x2b. The edges leading from invoked address node 0x0804842c to every return address node of the same memory region are shown in Figure 4-2.

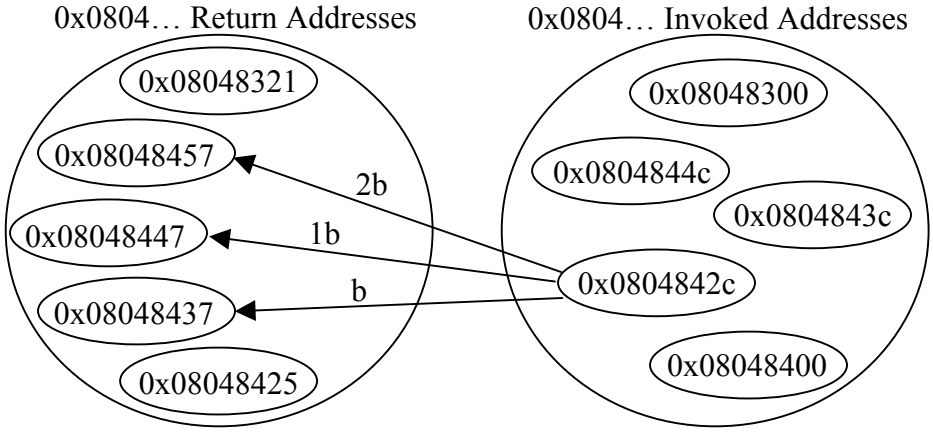

Figure 4-2. Edges leading from invoked address node 0x08048418.

Note that the node 0x0804842c was not given an edge to nodes 0x08048321 and 0x08048425. This is because these edges would have resulted in negative weights which we do not allow in the graph. This concept is explained more thoroughly in the next subsection.

Once all of the appropriate edges have been added, the graph is complete. We have omitted the drawing of our completed graph due to the crowded nature of a graph of such a simple program.

Graph Construction Explained

Inspection of Table 4-2 allows us to see a relationship between a value in the i^{th} row of column one and the $(i-1)^{th}$ row of column two. For example, we know that 0x0804842c is the address of the first instruction in our *f2()* function. Let the row with this address be the $(i-1)^{th}$ row. This means that the return address in the i^{th} row is 0x08048437. Since we also know that the last instruction in our *f2()* function is at

address 0x08048439, we know that this return address is inside of *f2()*. Furthermore, we can see in Figure 4-2 that when a minimum weight edge leaving the address node of 0x0804842c is chosen, it leads to the return address node of 0x08048437. Stated more formally, if we let return addresses be denoted with ω, and invoked addresses be denoted with α, a given invoked address, α_i, should have a minimum weight edge leading to return address, ω_{i+1}. This leads to the idea that every graph's GHCP is no different from the Actual Call Path (ACP) of the program.

It turns out, this is exactly what we need. All programs that have not fallen victim to a stack-smashing or buffer overflow attack posses this ACP. We can find this ACP by searching for a GHCP. Our method must be greedy to insure that we chose the minimum weight edge when leaving a given node. In addition, since our path must include each vertex exactly once, our path is Hamiltonian. If we are unable to find such a GHCP, then we know that our ACP has been disrupted. This implies the likely occurrence of a stack-smashing or buffer overflow attack.

To demonstrate why this works, suppose the function *f2()* were vulnerable to a buffer overflow attack. Suppose the attack overwrites the return address of *f2()* with the address 0x0804844a. This results in the edge from Figure 4-2 that was labeled with 0xb, now being labeled with a 0x1e. Thus when a minimum weight edge leaving the address node of 0x0804842c is chosen, it no longer leads to the proper node. It leads to the address node 0x08048447, whose edge is labeled with a 0x1b. This same address node is also the result of choosing a minimum weight edge when leaving the address node of 0x0804843c. Having two edges that both lead to the same node disrupts our GHCP. There no longer exists a path that is both Greedy and Hamiltonian. When the lack of a

GHCP is detected, we know that a stack-smashing or buffer overflow attack has occurred.

One assumption we make is that two functions in memory never overlap and that the initial instruction of a function is always the invoked address. We realize that some programs written in assembly may not abide by this assumption. However, all compiled programs and most assembly programs satisfy this constraint.

To summarize, our ACP represents the expected GHCP. However, we provide multiple paths leaving a given invoked address to give that invoked address a choice when determining our GHCP. By providing a choice, it allows the other return addresses to act as upper bounds. The upper bounds created by other return addresses limit the potential range of addresses that a given return address can been overwritten and modified to by an attacker. There already exists an inherent lower bound since we do not include negative weight edges. Recall that invoked addresses are likely the address of the first instruction for a given function. Thus it makes sense that unaltered execution flow of a given function should never lead to an instruction that resides at a lower memory address than the first instruction of that function.

Proof by Induction

In order for us to rely on the nonexistence of a GHCP to indicate the presence of a stack-smashing attack we must first prove that a GHCP exists for all uncompromised programs. In this section, we consider the case in which there are no recursive function calls. Knowing that our graph has two types of edges, those leaving return addresses and those leaving invoked addresses, we can simplify this proof. Since there is always exactly one edge leaving a given return address, we know this edge is always part of our GHCP. We can exploit this feature of our graph to simplify our proof. With this feature,

we now only need to prove that in the ACP each invoked address always has a minimum weight edge leading to its corresponding return address. We prove, using induction that this holds true for all unobjectionable programs. An unobjectionable program is defined as a program whose call stack represents a possible actual call path. Our formal inductive hypothesis is as follows:

Theorem 4.1: *For all unobjectionable programs in which n* different *functions have been called, where* $n \geq 1$*, every invoked address* α_i*, for* $i < n$*, has a minimum weight edge leading to the return address* ω_{i+1}.

With this stated we must first prove our base case.

Base case. In this case there is one active function and no other calls have been made. Our assertion that α_i, for $i < n$, has a minimum weight edge leading to return address ω_{i+1}, is vacuously true. Alternatively, we can say that the GHCP corresponds to the ACP, because they are both null.

Inductive case. For our inductive case we must prove that if the GHCP corresponds to an ACP for n calls, it corresponds to an ACP when the $(n+1)^{st}$ call is made. Stated more formally, we assume the following to be true:

$$\text{GHCP}_n = \text{ACP}_n = \alpha_1, \omega_2, \alpha_2, \omega_3, \alpha_3, \omega_4 \ldots \omega_n, \alpha_n$$

Thus we must prove the following to be true:

$$\text{GHCP}_{n+1} = \text{ACP}_{n+1} = \alpha_1, \omega_2, \alpha_2, \omega_3, \alpha_3, \omega_4 \ldots \omega_n, \alpha_n, \omega_{n+1}, \alpha_{n+1}$$

The $(n+1)^{st}$ call results in adding the two additional nodes, ω_{n+1} and α_{n+1}, to our graph. This also results in the additional edges, (α_i, ω_{n+1}) and (α_n, ω_{i+1}), being added to our graph. Since we know that GHCP = ACP, as long as every invoked address α_i, has a

minimum weight edge leading to the it's corresponding return address ω_{i+1}, we must prove the following proposition.

Proposition: *For each i, where i < n, weight*(α_i, ω_{i+1}) < *weight*(α_i, ω_{n+1})

Before proceeding any further we must define some variables.

Table 4-3. Variables for the induction proof.

Variable	Definition
L_i	Length in bytes of the i^{th} function.
α_i	Address of the first byte of the i^{th} function.
$r\alpha_i$	The offset to the return address inside the i^{th} function. ($r\alpha_i = \omega_{i+1} - \alpha_i$)

We also assume that two separate functions loaded into memory never overlap. Therefore, we must prove our proposition for two different scenarios, namely $\alpha_i < \alpha_n$ and $\alpha_n < \alpha_i$.

We can construct an abstract version of our graph as it would exist the moment our $(n+1)^{st}$ call is made. This version of our graph, Figure 4-3, illustrates the relationship between the function that made the $(n+1)^{st}$ call and any other invoked/return address pairs. A solid line represents an existing edge. A dotted line represents a new edge.

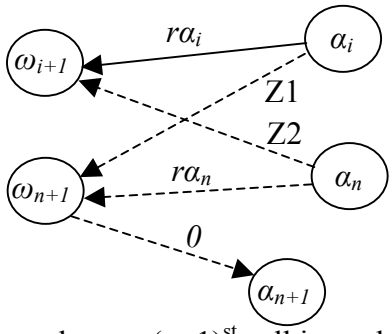

Figure 4-3. Abstract graph once $(n+1)^{st}$ call is made.

Now we prove our proposition holds for both scenarios. For the scenario

$$\alpha_i < \alpha_n,$$

we know the following must also be true

$$\omega_{i+1} < \omega_{n+1}.$$

Therefore we can conclude

$$weight(\alpha_i, \omega_{i+1}) = r\alpha_i = \omega_{i+1} - \alpha_i < \omega_{n+1} - \alpha_i = Z1 = weight(\alpha_i, \omega_{n+1}).$$

Thus our proposition holds true for our first scenario. Given the second scenario

$$\alpha_n < \alpha_i,$$

we know the following must also be true

$$\omega_{n+1} < \omega_{i+1}.$$

Therefore we can conclude

$$weight(\alpha_i, \omega_{n+1}) < 0,$$

and since our graph does not contain negative edges, our proposition still holds true.

It might seem logical to conclude that we also need to prove a second proposition. This second proposition is stated below.

Proposition 2: *For each i, where* $i < n$, $weight(\alpha_n, \omega_{n+1}) < weight(\alpha_n, \omega_{i+1})$

Proving this proposition for the first scenario we find

$$weight(\alpha_n, \omega_{i+1}) < 0.$$

Once again, since our graph does not contain negative edges, our proposition still holds true. With the second scenario we find

$$weight(\alpha_n, \omega_{n+1}) = r\alpha_n = \omega_{n+1} - \alpha_n < \omega_{i+1} - \alpha_n = Z2 = weight(\alpha_n, \omega_{i+1})$$

Thus we can prove that our 2nd proposition also holds for both scenarios. However, if Proposition 1 holds, then this second proposition is unnecessary. When we arrive at the point where we must choose an edge leaving α_n, since we are searching for a GHCP, our only feasible choice is $r\alpha_n$ leading to ω_{n+1}. If the first proposition holds, every ω_{i+1} for $i < n$, has already been visited. Thus the only choice that maintains a Hamiltonian path is ω_{n+1}.

In conclusion, we have proven that when the $(n+1)^{st}$ call is made, every invoked address α_i, still has a minimum weight edge leading to it's corresponding return address ω_{i+1}. This in turn proves that if the GHCP corresponds to the ACP for n calls, it corresponds to the ACP when the $(n+1)^{st}$ call is made. Therefore we know that the lack of a GHCP demonstrates that some form of stack-smashing or buffer overflow attack has occurred.

Recursion

Recursion is the case where $\alpha_i = \alpha_n$ for some $i < n$. When this is the case, we have two different scenarios that may create a problem.

- $\omega_{i+1} = \omega_{n+1}$
- $\omega_{i+1} > \omega_{n+1}$

The first scenario creates a problem because α_i has two equal weight edges leading to ω_{i+1} and ω_{n+1}. Subsequently, these two equal weight edges are also the minimum weight edges leaving α_i. When searching for a GHCP, we won't know which edge to choose. The second scenario creates a problem because α_i has a minimum weight edge leading to ω_{n+1}. To address these scenarios we add a new graph construction rule.

Rule 1: *If* $\alpha_i = \alpha_n$ *for some i, where* $i < n$*, we don't allow the edge* (α_i, ω_{n+1}) *in our graph.*

With this rule being stated, we must now prove that our GHCP corresponds to our ACP with this condition even when recursion is present. We now revisit each case of our induction proof in the previous section dropping the requirement that all active functions are different from each other.

It is important to note that we are not concerned about the scenario where $\omega_{i+1} < \omega_{n+1}$, for the same reasons we were not concerned about the Proposition 2 in Theorem 4.1.

We now prove the following theorem.

Theorem 4.2: *For all unobjectionable programs in which n functions have been called, where n ≥ 1, every invoked address α_i, for i < n, has a minimum weight edge leading to the return address ω_{i+1}.*

Base case (n = 1). Since this case has only one active function, the new condition has no affect on it. Once again, the GHCP corresponds to the ACP, because they are both null.

Inductive case (n > 1). For our inductive case we must prove that if the GHCP corresponds to the ACP for *n* calls, it corresponds to the ACP when the $(n+1)^{st}$ call is made even when our new condition is applied. We know that $GHCP_n$ still corresponds to ACP_n. We know this because before the $(n+1)^{st}$ call is made, α_n is the last node in our ACP_n. Hence, ω_{n+1} does not exist yet and neither of our scenarios create a problem yet.

Once the $(n+1)^{st}$ call is made, we must still prove that when our additional condition is followed that $GHCP_{n+1}$ corresponds to ACP_{n+1}. Fortunately, we know the following:

$$If\ i < n,\ then\ i \neq n$$

Thus,

$$(\alpha_i, \omega_{n+1}) \neq (\alpha_i, \omega_{i+1})$$

Since the edge (α_i, ω_{i+1}) is never the same edge as (α_i, ω_{n+1}) we can safely remove (α_i, ω_{n+1}) from our graph and our GHCP is not affected. Since (α_i, ω_{i+1}) is always left

unmodified, we know that our GHCP still exists. Figure 4-4 shows our abstract graph when the $(n+1)^{st}$ call is made for when $\alpha_i \neq \alpha_n$ and $\alpha_i = \alpha_n$.

Figure 4-4 illustrates that when the $(n+1)^{st}$ call is made, regardless of whether $\alpha_i \neq \alpha_n$ or $\alpha_i = \alpha_n$, $GHCP_{n+1}$ still corresponds to the ACP_{n+1}. Our new condition never alters our (α_i, ω_{i+1}) edges. In Figure 4-4, the left sides represents when $\alpha_i \neq \alpha_n$, and the right side represents when $\alpha_i = \alpha_n$.

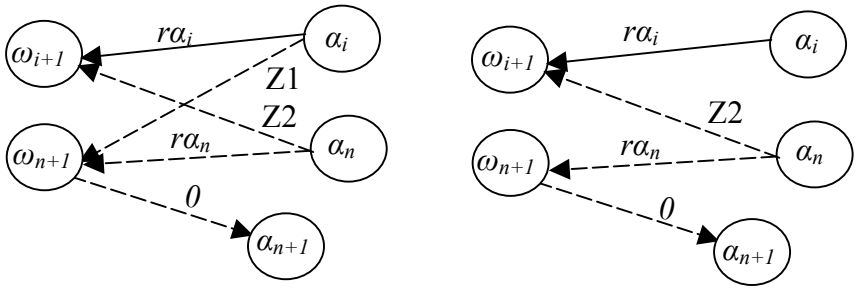

Figure 4-4. Abstract graph with recursion when the $(n+1)^{st}$ call is made.

To summarize, we use other function's return addresses to perform bounds checking on a specific function, say *f()*. We do not allow multiple invocations of *f()* to create bounds criteria for itself.

Implementation and Testing

We have implemented our method of detecting stack-smashing attacks in a prototype called Edossa (Efficient Detection Of Stack-smashing Attacks). Edossa was designed as a kernel module on the Linux 2.4.19 kernel. It was implemented on a system running RedHat Linux 7.3 with a 1.3 GHz AMD Duron processor. The system is running with 128MB RAM. The source code for Edossa is freely available under the GPL at http://www.cise.ufl.edu/~mfoster/research/edossa.html.

Edossa operates by checking the integrity of the program call stack at the point when a system call is made. As an initial proof of concept version, the current Edossa is

only intercepting the *execve()* system call. While there are numerous systems calls that could be used maliciously, the vast majority of buffer overflow and stack-smashing attacks deal with the attacker attempting to spawn a root shell. This is done by passing the string "/bin/sh" to the *execve()* system call. The intercepting of additional system calls could be easily added to Edossa with minimal effort.

To test Edossa we collected a number of publicly available exploits. These exploits consisted of a wide range of applications that are known to have some form of buffer overflow vulnerability in their source code. The results of our testing are shown below in Table 4-4.

Table 4-4. Publicly available exploits used to test Edossa.

Exploit	Date Posted	Result	Overhead (μs)	Reference
efstool	Dec. 02	Attack Detected	94	[9], [10]
finger	Dec. 02	Attack Detected	104	[9]
gawk	April 02	Attack Detected	113	[9]
gnuan	July 03	Attack Detected	126	[9]
gnuchess	July 03	Attack Detected	103	[9]
ifenslave	April 03	Attack Detected	94	[10], [11]
joe	Aug. 03	Attack Detected	90	[9]
nullhttpd	Sept. 02	Attack Detected	103	[9]
pwck	Sept. 02	Attack Detected	212	[9]
rsync	Feb. 04	Attack Detected	109	[9]

As we can see from Table 4-4, Edossa was successful at detecting all of these known exploits. In addition, the overhead for detecting all of these exploits never exceeded 212 microseconds of CPU time.

Limitations

One limitation of our GHCP analysis is that it depends on the existence of a valid frame pointer. In most cases when the return address is overwritten, the frame pointer is also overwritten. Without a valid frame pointer, there is no way to trace through the stack to extract return addresses. However, in the case where there is no valid frame

pointer, we already know that some form of buffer overflow or stack-smashing attack is underway. A system that implements our proposed method, such as Edossa, detects that no valid frame pointer exists even before generating a graph. A system that only tests for the ability to trace up a stack is too easily evaded by an attacker to warrant a stand alone buffer overflow detection system. However, due to the prevalence of attacks that could be detected with such a test, we believe it should be incorporated.

There are two methods with which an attacker might be able to evade detection of a system using GHCP analysis. The first method an attacker could use to evade detection is to perform a buffer overflow attack that overwrites a return address to a new return address but leaves the frame pointer unmodified. The second method an attacker could use to evade detection, is to perform a buffer overflow attack that overwrites the return address and frame pointer and also injects code onto the stack. The first few instructions of this injected code must restore the return address and frame pointer to their original values. The injected code would then jump to preexisting code the attacker wants to execute. Both of these methods could work, but they exhibit a major limitation. The new return address or the preexisting code jumped to by the injected code must reside in the same function as the original return address. Recall that each invoked address must have a minimum weight edge to its corresponding return address. If the new return address is inside of another function, the attacker risks destroying the minimum weight edge between the invoked address and the original return address. Likewise for the second method. When a system call is made, a return address is placed on the stack. Thus, if the injected code has jumped to a piece of code in another function, the attacker risks this return address not having a minimum weight edge from its corresponding invoked

address. While both methods are possible, the challenges facing the attacker are much more rigorous than without GHCP analysis.

Another limitation of our method is its ability to deal with function pointers. Currently, we use return addresses to trace through memory and find a corresponding invoked address. These invoked addresses are part of a *call* instruction in memory. The bytes in memory representing a *call* instruction include an address or offset to an address. In either case we can extract the address invoked by a *call* instruction. In the case of function pointers, the *call* instruction is often calling an address that is in a register. We have no way to determine what address was in this register when this *call* instruction executed. However, we note that one can easily modify a compiler to store invoked addresses on the stack. We have done this in the form of a patch for the GCC compiler to verify the technique. This gives us the ability to always determine a return address' corresponding invoked address. When using programs compiled with the patched GCC compiler, function pointers are no longer a limitation.

Benefits

A system designed for buffer overflow detection using GHCP analysis has a number of benefits. First, our system does not require access to a program's source code. Secondly, it does not require that a program be compiled with any specially enhanced compiler or even have the executable binary file rewritten unless one wants to verify calls through pointers to functions. In addition, our system does not require linking with any special libraries or place any additional burden on the application programmer. Many intrusion detection systems also rely on a training phase with a program to learn its normal behavior. After a training phase, the system can monitor a program to ensure that

it doesn't deviate from the behavior observed during the training. Our system does not require any training phase.

Our method is similar to a nonexecutable stack because it makes it extremely difficult for an attacker to execute malicious code on the stack. However, our method provides a number of benefits the executable stack does not. For example, in addition to stack-smashing attacks, our method can also detect heap smashing attacks. Furthermore, it is likely to also detect a similar attack that uses the bss or data segment. Our method would also detect most attempts to rewrite a return address to another location in preexisting code. A nonexecutable stack would not detect such an attack. Lastly, our method allows code with a legitimate stack trace to execute code on the stack. In cases where an uncompromised process needs to execute code on the stack, the nonexecutable stack would not allow such a process to proceed.

Our method also provides the framework for even more concise buffer overflow detection system. Currently, one of the limitations of our method is that we rely on other functions in the call path for our bounds checking criteria for a given function. A compiler could easily be modified to inject a dummy function in between every function in a given program. The code for the i^{th} dummy function would consist of only the code required to call the $(i+1)^{st}$ dummy function. By calling the sequence of dummy functions before starting *main*() we would place the necessary bounds checking criteria on the stack that we need for any function in our program. The cost of this is in the compilation and start up times of the program. In addition, computation of the GHCP would only require time proportional to the number of active functions.

CHAPTER 5
PROCESS FORENSICS

Proposed Process Forensics

All work done on a computer system is done in the form of a process. Processes can be divided into two categories: user-space processes and kernel-space processes. For the purpose of this discussion we refer only to user-space processes. The reason for this is that a given kernel-space process is always acting on behalf of a particular user-space process or processes. Regardless of the unique methods different platforms use to handle processes, most all processes contain a great deal of information. Unfortunately, due to the nature of computer forensics, by time a forensic investigation has begun, most of the relevant processes have already been terminated. Often, the involved computer system may have been completely shutdown. The only data with which a digital forensic investigator has to analyze processes are any log files created while a given process was executing. Thus, the digital forensic investigator is left with a very limited amount of computer forensics concerning processes. We believe computer forensics can and should be expanded to include more process information. We propose checkpointing as one means to create more computer forensics in the form of process forensics.

Knowing the importance of log files, let us analyze the sources of log files. Essentially, logging is the same as recording details about the currently running processes. In other words, logging is no more than creating nonvolatile data by recording details about volatile data. This perspective immediately leads us to consider what other volatile data should be made nonvolatile. Knowing that a significant portion of volatile

data comprises user processes, we see where checkpointing can be a means to create more nonvolatile data out of volatile data.

Possible Evidence in a Checkpoint

Let us continue our discussion with a quick overview of the main sources of information found in the checkpoint of a process. Recall, a process exists in main memory where it has been assigned its own address space. Some of the more useful information found in a process's address space, therefore included in a checkpoint, might consist of items such as a process identification (PID), the user who owns the given process, and pointers to parent, child, and sibling processes. While an attacker may have altered some of this information, it still provides a starting point for the forensic investigator. Information such as a PID is essential in distinguishing between multiple processes. Furthermore, knowing what user owned a process indicates who started the process or whose account has been compromised. Ownership of a process, whether legitimate or not, also tells us the permissions level of the process. Clearly, a process run as root can do far more damage than a typical user process. In addition, knowing the relationship between different processes can assist in isolating the source of a process or what other processes resulted from the execution of a process. The parent and sibling relationship between processes is something not likely found in log files.

One of the more notable portions of a process's address space is the stack. The stack contains significant information pertaining to the execution sequence of a process. This sort of information is extremely useful to someone investigating a buffer overflow or *stack-smashing* attack [31]. Given access to the stack, a digital forensics investigator can determine both where and how such an attack was made possible. Without knowing where and how an attack was made possible, it is very difficult to prevent similar future

attacks without limiting one's usage of their own system. The process address space also contains the heap, bss and data segments of a process. Analysis of the heap segment may reveal evidence pertaining to a *heap smashing* attack much like the stack in a buffer overflow or stack-smashing attack. The stack, heap, bss, and data segments are all potential targets of malicious input attacks. In turn, each of these items would contain essential evidence of such an attack. As an integral part of the process address space, each of these items is included in a checkpoint image file. An additional example of this sort of malicious input attack would be a *format string attack*.

A process' address space also contains information about items we refer to as process peripherals. Process peripherals include opened files, sockets, and pipes. Knowing what files a process accessed can be extremely valuable to the forensic investigator. This can indicate the intruder's objective, help isolate the damage done during the attack, or indicate attempts by the intruder to cover his or her own tracks. The digital forensic investigator and system administrator very much need to know if files such as password or log files have been modified or accessed. Tampering with a password file indicates the likelihood of future attacks via a compromised account. When log files have been tampered with it usually indicates an attacker is attempting to cover his/her tracks. Furthermore, socket connections provide additional evidence of communication links involved in a crime. Socket connections may indicate from where an attacker is launching an attack or where the attack is dumping stolen data. Pipes are another form of communication with which that digital forensic investigator would take an interest. Some checkpoints even include data that is still in the pipe buffer. Process peripherals could also include items such as a process' corresponding tty or terminal.

With this information, the digital forensic investigator might learn the attack was launched locally.

The possibilities of evidence from process forensics are quite vast. However, to gain from process forensics we must know when to collect process forensics. The following section addresses this issue.

Opportunities for Checkpointing

A system administrator must make a number of tough decisions when dealing with an intruder. At times, a system administrator may become aware of an attack while the attack is in progress. This may be a result of the system administrator's own monitoring of the system or an alert issued by an Intrusion Detection System (IDS). The knee-jerk reaction to such a scenario is to kill all the processes related to the attack. While such an approach can be very effective in stopping the attack, it does little towards collecting evidence about the attack. Furthermore, such an approach is likely to tell the intruder that he or she was detected. Most of the time, one does not want the intruder to know he or she was detected, until there is enough evidence to prove a crime took place, and by whom it was committed. We believe when an intrusion is detected, whether by the system administrator or IDS, the immediate actions should include collection of evidence, or more specifically process forensics. We encourage the use of incremental checkpoints that can be created without alerting the intruder. Once the intruder's session is ended, whether by the system administrator or by the intruder himself, the resulting checkpoints can provide crucial information about the attack.

A recent look at the ICAT vulnerability statistics shows a significant number of the CVE and CVE candidate vulnerabilities were due to buffer overflows. For the years 2001, 2002 and 2003 buffer overflows accounted for 21, 22, and 23% of the

vulnerabilities, respectively [6]. While much work has been done to detect buffer overflow attacks, to the knowledge of this author, little has been done to enhance our abilities to collect evidence resulting from buffer overflow attacks. We believe process forensics derived from checkpointing can help fill this void. Recall that a checkpoint contains the stack, heap, data, and bss segments of a process. In the case of a buffer overflow attack, creating checkpoints the moment the attack is detected, and even while the attack is in progress, is likely to collect vital evidence. A forensic investigator can use this information to more closely determine how and when the intruder entered the system. A thorough analysis of the stack is likely to show what function contains the exploited vulnerability. Isolating the vulnerability is essential to preventing a similar attack in the future. Furthermore, in the case of a stack-smashing attack, any code injected onto the stack may uniquely correspond to code that is later found on the attacker's computer. Likewise for a heap smashing attack and a process's corresponding heap segment. While this alone does not prove anything, it does provide an additional corroborating stream of evidence. Any additional such evidence is desirable in the case of a legal setting. Stephenson [26] reminds us that it takes a "heap of evidence, to make one small proof."

ICAT's CVE and CVE candidate vulnerabilities classified as buffer overflow attacks are actually a subgroup of a much larger classification. This larger classification, known as *input validation errors*, accounted for 49%, 51%, and 52% of the CVE and CVE candidate vulnerabilities for the years 2001, 2002, and 2003 respectively. The idea of collecting evidence about a buffer overflow attack from a checkpoint is based on the concept that a buffer overflow attack stems from malicious input. Such input has no

choice but to become part of a process's address space. We believe this approach to process forensics and evidence collection can be expanded far beyond buffer overflow attacks to include other input validation errors. An example of an input validation error is a boundary condition error. While some boundary condition errors result from a system running out of memory, others may result from a variable exceeding an assumed boundary. Inspection of variables in a checkpoint file may reveal such an assumed boundary and expedite the process of closing a vulnerability once exposed. SQL injection attacks also take advantage of input validation errors. SQL injection attacks often allow the attacker to damage and/or compromise a website's database. The very nature of attacks that exploit input validation errors automatically leave evidence in a process's address space. The potential for evidence and process forensics from checkpointing intruder related processes resulting from such vulnerabilities have yet to be explored.

Most intrusion detection systems can be categorized as misuse detection and anomaly detection. Misuse detection usually refers to those systems that utilize some form of signature or pattern matching to determine whether or not a process is part of an intrusion. Anomaly detection usually refers to those systems that attempt to define *normal* behavior so that processes can be categorized as normal or intrusive. Due to the inherent challenge in defining what is *normal* behavior, these systems often rely on some form of threshold to distinguish between normal and anomalous behavior. Markov Chain Model [32], Chi-square Statistical Profiling [33], and Text Categorization [34] are examples of such approaches to anomaly detection. We propose that such anomaly detection systems use checkpointing as an evidence collection technique for processes

that are approaching or have passed the given threshold. Incremental checkpoints can be used to continually collect evidence of a process's behavior for any process that is considered anomalous or nearing anomalous. This would result in process forensics for those malicious processes that never quite reach the threshold and would usually go undetected. In addition, this would create process forensics for processes that do cross the threshold. Such forensics could expedite finding out why a process deviated from its normal behavior.

A common dilemma facing the computer crime investigator when entering a crime scene is whether or not to unplug the computer [28]. Any work by the criminal that resides in main memory is lost if the computer is unplugged. However, forensic analysis of a hard disk must always be performed on a copy rather than the original. In order to create a copy of the confiscated hard disk, the computer must eventually be powered off. Depending on the platform, Stephenson usually recommends directly unplugging the power source [26]. This avoids any booby-traps that may be triggered if the machine is not shutdown in a particular manner. Regardless of the manner by which a machine is shutdown, all of the volatile data such as a running process is lost. This illustrates another example of where additional evidence may be gained by using checkpointing. Prior to shutting down or unplugging a computer, relevant processes could be checkpointed. The resulting checkpoint files allow the forensic investigator to analyze the running processes at a later time.

Additional Enhancements

If a computer crime ever reaches the courtroom, any evidence presented before the court must have been preserved through a chain-of-custody [26]. In other words, one must be able to verify with whom and where the evidence has been held since the

moment it was collected. In the case of a checkpoint, the checkpoint resides in a file and can therefore be digitally fingerprinted immediately following its creation. In a courtroom, this digital fingerprint can be verified to show that the checkpoint file remains unaltered. A time and date stamp can also be included and verified with a digitally fingerprinted checkpoint file.

In addition, a checkpoint stored as a file can easily be transferred to a secure location much like some logging systems. It is often recommended that logs be stored on a separate secure system from the system that generates the logs. These log files are also commonly stored in an encrypted format. These measures deter an intruder from altering log files to cover-up his or her unauthorized access to a system. Checkpoint files can be treated in the same manner. They can be stored on secure systems separate from where they were created. This prevents an intruder from modifying or destroying any evidence that is collected in the form of a checkpoint file.

Sommer has provided a good analysis of why intrusion detection systems fall short of providing quality evidence in [29]. We propose that a checkpointing system should be developed separately from an ID system. During an attack, the ID system can trigger the checkpointing system to handle any intrusion related processes. This allows the ID research to focus on detection rather than evidence collection. A checkpointing system, due to its inherent goal of recreating a process, is already aimed at collecting information about a process. We believe this goal can be more easily combined with the goal of evidence collection. Furthermore, by allowing checkpointing systems to provide the evidence collection, we alleviate the need for drastic modifications to existing ID systems.

We also believe the format of a checkpoint file could be shared amongst multiple platforms. We believe that the format of a checkpoint file should be standardized similar to that of the ELF format used on Linux platforms. The standardization of checkpoint file formats would facilitate a common ground from which law enforcement, academia, and other researchers can work. This would facilitate the development of tools for working with and analyzing checkpoint files. In addition, standardizing any aspect of the forensic investigation aids in training future forensic investigators. Furthermore, standardization would assist in the acceptance of checkpoint evidence in legal proceedings. Likewise, standardization could further facilitate process migration amongst different platforms.

Carrier [27] has proposed a balanced solution to the open/closed source debate with regards to digital forensic tools. Carrier urges digital forensic tools be categorized into tools for extraction and presentation. Carrier proposes extraction tools should remain open source, while presentation tools remain closed source. Such a balanced solution could easily be applied to checkpointing tools. The checkpoint/restart engine of a checkpointing system could remain open source. This allows researchers and the digital forensics community to validate the inner-workings of such checkpointing tools. Meanwhile, the presentation tools used for presenting the data from a checkpoint file could remain closed source. It is likely that many individuals involved in the legal proceedings following a computer crime do not posses the necessary technical skills for understanding the data found in a checkpoint file. This provides ample opportunity for software developers to create presentation tools for checkpoint files. The goal of making

complex checkpoint file data easily understandable would create ample competition for the private sector.

CHAPTER 6
CONCLUSIONS

In this paper we have shown significant progress towards developing the ideal checkpointing system. We define the ideal checkpointing system as one that satisfies the Three AP's of Checkpointing which are: Any Process on Any Platform at Any Point in time. The system we develop and discuss in this paper supports two of the three AP's: Any Process at Any Point in time. To achieve these two AP's, we developed UCLiK as a kernel module. As a kernel module we enjoy levels of transparency not experienced by most checkpointing systems. This transparency includes relieving the application programmer from any additional responsibility, not requiring any system call wrappers to log information about a process, no special compilers and no checkpointing libraries with which one must compile or link their programs. Additional transparency is achieved by developing the system as a kernel module since the running kernel's code does not have to be modified.

The benefits of such a checkpointing system are very broad. A checkpointing system such as UCLiK can be used for process migration, fault tolerance and rollback recovery. Furthermore, UCLiK can provide system administrators with an alternative to the *kill* system call. System administrators who substitute *kill* with checkpointing a process, now have the option to undo ending a given process' execution. This functionality allows system administrator's to more preemptively protect their systems against runaway and other suspicious processes without losing valuable work.

Our study has also served to deepen our belief that development of a single checkpointing system that can satisfy the third AP to checkpointing, namely Any Platform, is unrealistic. However, to best facilitate the long term goal of checkpointing processes on *Any Platform* we suggest development of checkpointing systems be divided into two separate components. The two components are the checkpoint engine and the checkpoint file. We refer to the checkpoint engine as the portion of the system concerned with collecting process data, kernel state, and any other interactions with the operating system. This component is also concerned with stopping and restarting a process. The checkpoint file is the actual file where a checkpoint image is saved and can be stored indefinitely. We believe differences in hardware architecture and kernel data structures make it nearly impossible for a single checkpoint engine to work on all existing platforms. Thus, the development of checkpoint engines for different platforms are probably best kept separate. However, we suggest that the development of checkpoint engines for different platforms should be coordinated. This coordination should be centered around designing the engines to work with a standardized checkpoint file format. We believe checkpoint file formats should be standardized in a way similar to that of ELF files. Standardizing the checkpoint file format would have numerous benefits. One major benefit would be facilitating cross-platform migration. Checkpoint engines of different platforms could all function on the same checkpoint file. Furthermore, tools for reading and modifying the contents of a checkpoint file could be developed and would immediately be platform independent. Additional benefits to a standardized checkpoint file format with regards to computer forensics are prevalent and discussed shortly.

In this paper we have also introduced a novel method for detecting stack-smashing and buffer overflow attacks. We have shown how the return addresses extracted from the program call stack can be used along with their corresponding invoked addresses to create a weighted directed graph. We have shown that such a graph always contains a Greedy Hamiltonian Call Path (GHCP) for all unobjectionable programs. Thus, the lack of a GHCP can be used to indicate the existence of a stack-smashing or buffer overflow attack. The benefits of such a method for detection are independence from specially enhanced compilers and libraries, access to a program's source code is unnecessary, executables do not have to be rewritten, there is no logging involved and it requires no modifications to the operating system. These benefits make our approach unique when compared with most other approaches to detecting stack-smashing and buffer overflow attacks.

In addition, our work has laid the framework for an even more concise detection system for stack-smashing and buffer overflow attacks. Using our methods in addition to an enhanced compiler could remove the limitations experienced by our system involving function pointers and programs with few active functions. An existing compiler could be easily modified to include a series of dummy functions that are called at the beginning of a program's execution with the sole purpose of placing bounds checking criteria on the stack. Furthermore, enhancing a compiler to push the invoked addresses on the stack would allow our method to handle function pointers.

We have begun implementing a prototype for our method. Early results show a promising outlook for low overhead. Future work includes continued development of our

prototype with more exhaustive testing of overhead and compatibility with items such as setjmp/longjmp calls. We expect such items to be compatible with our method but it remains unconfirmed and beyond the scope of this paper.

Researching both checkpointing and intrusion detection results in a unique perspective, namely, that computer forensics is lacking in the subfield we have termed process forensics. Process forensics involves extracting information from a process address space of a given program for the purpose of evidence collection. Since computer forensics is restricted to nonvolatile data, to improve computer forensics we must find new sources of nonvolatile data. Since checkpointing creates nonvolatile data from processes, we believe that including checkpointing technology with intrusion detection systems can create a new source of nonvolatile data. In addition, by increasing the amount of nonvolatile data we have increased the amount of forensic evidence available to the digital forensic investigator. Since this evidence comes from processes, we find it appropriate to refer to it as process forensics. In this paper we have explored different sources and benefits of process forensics. One primary example being the evidence collected by an intrusion detection system enhanced with checkpointing technology.

Palmer [35] reminds us that the future is likely to bring even tougher standards for digital evidence. We believe standardizing items such as the checkpoint file format used for process forensics can help meet these standards. Standardizing methods of evidence collection can help thwart some of the scrutiny placed on digital evidence in a courtroom setting. In addition, standardizing the checkpoint file format helps facilitate the training process of future digital forensic investigators. Lastly, it encourages the development of tools used to analyze checkpoint files for the purpose of process forensics.

In [26], Stephenson addresses the importance of reconstructing the crime scene. We suggest that anything less than the ability to recreate an entire process state may lead to holes in the evidence required to identify an attacker or prevent similar future attacks. Checkpointing provides us with the necessary level of detail to recreate an entire process. Although many attacks to date have not necessitated checkpointing, we do not want to use the needs of the past to limit our preparedness for the future.

In closing, researchers and the digital forensics community must continue to find new sources of evidence following computer crimes. We believe that in many cases checkpointing technology can achieve such a goal.

LIST OF REFERENCES

1. National Telecommunications and Information Administration, Economics and Statistics Administration, "Nation Online: How Americans Are Expanding Their Use of the Internet," Author, Washington, D.C., February, 2002, Retrieved May 10, 2004, From http://www.ntia.doc.gov/ntiahome/dn/.

2. Carnegie Mellon University, "CERT Coordination Center Statistics," Author, 2004, Retrieved May 10, 2004, From http://www.cert.org/stats/cert_stats.html.

3. A.B. Brown, D.A. Patterson, "Rewind, Repair, Replay: Three R's to Dependability," 10th ACM SIGOPS European Workshop, Saint-Emilion, France, September, 2002.

4. A.B. Brown, D.A. Patterson, "To Err is Human," Proceedings of the First Workshop on Evaluating and Architecting System Dependability, EASY '01, Göteborg, Sweden, July, 2001.

5. D.Wagner, J. Foster, E. Brewer, A. Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," In Proceedings 7th Network and Distributed System Security Symposium, Feb, 2000.

6. National Institute of Standards and Technology (NIST), "ICAT Vulnerability Statistics," Author, September, 2003, Retrieved April 2, 2004, From http://icat.nist.gov/icat.cfm?function=statistics.

7. SecurityGlobal.net, "SecurityTracker.com Statistics," Author, 2002, Retrieved April 2, 2004, From http://www.securitytracker.com/learn/securitytracker-stats-2002.pdf.

8. J. Plank, M. Beck, G. Kingsley, "Libckpt: Transparent Checkpointing under Unix," Conference Proceedings, USENIX Winter 1995 Technical Conference, New Orleans, LA, January, 1995.

9. J. Plank, Y. Chen, K. Li, M. Beck, G. Kingsley, "Memory Exclusion: Optimizing the Performance of Checkpointing Systems," Software -- Practice and Experience, Vol. 29, Number 2, pp. 125-142, 1999.

10. M. Elnozahy, L. Alvisi, Y.M. Wang, D. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," ACM Computing Surveys, Vol. 34, Issue 3, pp. 375-408, September, 2002.

11. L.M. Silva, J.G. Silva, “System-Level versus User-Defined Checkpointing,” Proceedings of the Seventeenth IEEE Symposium on Reliable Distributed Systems, pp. 68-74, October, 1998.

12. M. Litzkow, T. Tannenbaum, J. Basney, M. Livny, “Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System,” Paper Retrieved June 1, 2003, From http://www.cs.wisc.edu/condor/doc/ckpt97.ps.

13. G. Stellner, “CoCheck: Checkpointing and Process Migration for MPI,” Proceedings of the 10th International Parallel Processing Symposium (IPPS '96), Honolulu, HI, April 15-19, 1996.

14. S.E. Choi, S.J. Deitz, “Compiler Support for Automatic Checkpointing,” Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications (HPCS’02), pp. 213-220, 2002.

15. M. Beck, G. Kingsley, J. Plank, “Compiler-Assisted Memory Exclusion for Fast Checkpointing,” IEEE Technical Committee on Operating Systems and Application Environments, Winter, 1995.

16. B. Ford, M. Hibler, J. Lepreau, P. Tullmann, “User-Level Checkpointing through Exportable Kernel State,” Proceedings of the 5th International Workshop of Object Orientation in Operating Systems (IWOODS ’96), Seattle, WA, October, 1996.

17. A. Barak, O. La’adan, “The MOSIX Multicomputer Operating System for High Performance Cluster Computing,” Journal of Future Generation Computer Systems, Vol. 13, Number 4-5, pp. 361-372, March, 1998.

18. Y. Zhang, J. Hu, “Checkpointing and Process Migration in Network Computing Environment,” Proceedings of the 2001 International Conferences on Info-tech and Info-net, Vol. 3, pp. 179-184, Beijing, October–Novermber, 2001.

19. E. Pinheiro, “Truly-Tansparent Checkpointing of Parallel Applications,” Author, 2003, Paper Retrieved June 1, 2003, From http://www.research.rutgers.edu/~edpin/epckpt/paper_html.

20. H. Zhong, J. Nieh, “CRAK: Linux Checkpoint/Restart As a Kernel Module,” Technical Report: CUCS-014-01, November, 2001, Paper Retrieved June 1, 2003, From http://www.ncl.cs.columbia.edu/research/migrate/crak.html.

21. C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Waggle, Q. Zhang, “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks,” Proceedings of the 7th USENIX Security Conference, San Antonio, TX, 1998.

22. A. Baratloo, N. Singh, “Transparent Run-Time Defense Against Stack-smashing Attacks,” In Proceedings of the 2000 USENIX Technical Conference, San Diego, CA, Jan 2003.

23. H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, W. Gong, "Anomaly Detection Using Call Stack Information," IEEE Symposium on Security and Privacy, Berkeley, CA, May, 2003.

24. M. Prasad, T. Chiueh, "A Binary Rewriting Defense Against Stack Based Buffer Overflow Attacks," In Proceedings of the 2003 USENIX Technical Conference, San Antonio, TX, June, 2003.

25. D. Wagner, D. Dean, "Intrusion Detection via Static Analysis," IEEE Symposium on Security and Privacy, Oakland, CA, 2001.

26. P. Stephenson, "Investigating Computer-Related Crime," CRC Press, 1999.

27. B. Carrier, "Open Source Digital Forensics Tools: The Legal Argument," @Stake Research Report, October, 2002, Retrieved May 20, 2004, From http://www.atstake.com/research/reports/acrobat/atstake_opensource_forensics.pdf.

28. A. Yasinsac, Y. Manzano, "Policies to Enhance Computer and Network Forensics," Proceedings of the 2001 IEEE Workshop on Information Assurance and Security, West Point, NY, June, 2001.

29. P. Sommer, "Intrusion Detection Systems as Evidence," First International Workshop on the Recent Advances in Intrusion Detection, Belgium, September, 1998.

30. M. Foster, J.N. Wilson, "Pursuing the Three AP's to Checkpointing with UCLiK," Proceedings for the 10th International Linux System Technology Conference, October, 2003.

31. M. Foster, J.N. Wilson, S. Chen, "Using Greedy Hamiltonian Call Paths to Detect Stack-smashing Attacks," Proceedings of the 7th Information Security Conference, Palo Alto, CA, September, 2004.

32. N. Ye, "A Markov Chain Model of Temporal Behavior for Anomaly Detection," Proceedings of the 2000 IEEE Workshop on Information Assurance and Security, West Point, NY, June, 2000.

33. N. Ye, Q. Chen, S. M. Emran, K. Noh, "Chi-square Statistical Profiling for Anomaly Detection," Proceedings of the 2000 IEEE Workshop on Information Assurance and Security, West Point, NY, June, 2000.

34. Y. Liao, V. R. Vemuri, "Using Text Categorization Techniques for Intrusion Detection," 11th USENIX Security Symposium, August, 2002.

35. G.L. Palmer, "Forensic Analysis in the Digital World," International Journal of Digital Evidence, Volume 1, Issue 1, Spring, 2002.

BIOGRAPHICAL SKETCH

Mark Foster received the BS degree in Computer Science and Mathematics from Vanderbilt University in the Spring of 1999. Since the Fall of 1999, he has been a graduate assistant in the department of Computer and Information Science and Engineering at the University of Florida. He completed the MS degree in Fall of 2001. His academic interests include checkpointing, intrusion detection, and computer forensics. He also enjoys teaching, exercising, and movies.