# An Intrusion Prevention System using Wrapper *

Tsung-Yi Tsai, Kuang-Hung Cheng, Chi-Hung Chen, Wen-Nung Tsai
*Department of Computer Science and Information Engineering,*
*National Chiao-Tung University*
*{tytsai, chengkh, chihung, tsaiwn}@csie.nctu.edu.tw*

## *Abstract*

*Over the past several years, the Internet environment has become more complex and untrusted. There are always crackers and business competitors trying to penetrate security system and then steal confidential information. Some of them would also spread malicious software or files to attack our computer system, making our system paralyzed, unable to provide service. Even more, the attacker may gain full access to our system without any trace.*

*Based on the system call interception technique, we developed a real-time intrusion prevention system, IPSW (Intrusion Prevention System using Wrapper). This system intercepts every system call invoked by applications and tries to match any of the penetration scenarios. Once there is any evidence showing some penetration being undertaken, the system can terminate the penetration process before it gets injured. This wrapper system can also wrap Commercial Off-The-Shelf (COTS) software components to provide robustness and security.*

**Keywords:** IDS, IPS, System call Interception, Virus, Wrapper.

## 1. Introduction

Computer security has been a seesaw battle between users and intruders since long time ago. With the help of security tools like firewalls, anti-virus products and intrusion detection systems, it seems that the situation has been controlled. However, no matter how powerful these security tools are, there are always new tricks to elude their security defense line ingeniously.

Take anti-virus products as example, they use virus-definitions as the checking rules to detect whether a certain file is a virus. Virus-definition is a collection of some machine instructions and meta-data of this virus. These machine instructions and meta-data are chosen carefully so that the probability of another file having the same virus-definition is very small. Although anti-virus engines can detect suspect virus file accurately using pattern matching, they still cannot detect variant virus of the same family very well.

To stop hackers and to avoid the above situation, the computer security system should provide more abstract representation of attack behavior and more flexible configuration interface to absorb minor variations. For example, we know that a virus is a piece of code that copies itself into another program. By this definition, we can roughly say that: "Every process that opens an executable file and inserts itself into this file should be monitored". Certainly, this statement still needs some modifications to adapt to various virus infection techniques (like compression) and target platform, but it did provide a more general definition of all kinds of viruses than thousands of hundreds of virus signatures.

From another point of view, only when the intruder has accessing privilege to system resource can he cause certain degree of damages to the victim system. And the only possible way to access system resource is via the system call interfaces provided by the Operating System.

In this paper, we proposed a real-time Intrusion Prevention System (IPS) that can be implemented in the Linux kernel. Based on the state-transition diagram technique, we can define an attack pattern as a set of states and transition arcs. Each state represents one of the key signatures of this attack scenario and each arc connected with two states represents the required system call invocation that causes transition from one state to another state. With visual thinking, security officers can define attack patterns more easily and intuitively, and the attack pattern can be more complex than ever with less maintenance efforts.

To take a better battle position, the IPS will wrap around the kernel like a shell. The kernel process of IPS engine has the full information of target system. With this complete information and splendid strategy point, the IPS can check every new created process and determine whether this process is a malicious one. Once been judged suspected, the suspected process will be monitored all his life until terminated

* This work was supported in part by National Science Council, Taiwan, Contract No. NSC92-2213-E-009-096.

finally, or will be ferreted out as a compromise process during execution time before damage occurs.

This paper is organized as follows. Section 2 is a brief discussion of related work. The strategies we choose and the resulting system architecture are described in section 3, followed by the technical details of our approach in section 4. Section 5 gives two applied examples in our security model. Section 6 presents the conclusions and future works.

## 2. Related Work

Much of the work related to this paper falls in two categories: kernel-level wrapper [11][1][9] and state-based intrusion detection system [7][4][3].

### 2.1 Kernel-Level Wrapper

In order to provide accurate and efficient intrusion prevention, most wrapper systems will reside in Kernel space with superuser execution privilege. For efficiency reason, there will be no extra context switches compared with user-level process tracing technique. For accuracy reason, the kernel wrapper has equal rights as the kernel code and hence has full access to system information and system call parameters. In the following paragraphs, we will have a brief discussion for each kernel wrapper that is implemented in FreeBSD and Linux operating system respectively.

Generic Software Wrappers [11], or GSW in short, is a generic wrapper prototype developed by NAI labs for a Solaris or FreeBSD platform. GSW is a kernel-resident and non-bypassable software extension for augmenting security without modification of COTS source. The wrapper in GSW is a software module that surrounds other software components and is used to augment and control interactions between components. The GSW provides a Wrapper Life Cycle framework (WLC) to manage wrappers. WLC uses a small configurable rule-based database to manage the run-time relationships between wrapper instances and processes executing COTS applications. Every wrapper will experience five states in its life cycle: install, activate, duplicate, deactivate and uninstall.

Linux Kernel Loadable Wrappers [9], KLW in short, provide non-bypassable security wrappers for application specific security requirements and can also be used to provide replication service. The primary goal of KLW is to protect the user system from malicious active content downloaded via web browser. It develops three application specific wrappers: the Netscape KLW, the Apache Server KLW and the replication KLW. A Netscape KLW is used to protect a user from downloading and executing malicious active content when browsing on the Internet. An Apache Server KLW is used to restrict the web server to access only a subset of files. In this way, even if the web server is compromised, other resource that the Apache server cannot access will be secure. A replication KLW is used to replicate a file or set of files transparently. The replication KLW can backup of changes to a file immediately without having to modify any applications that are making the actual change.

### 2.2 STAT Intrusion Detection System

STAT [7][4][14] is the acronym of State Transition Analysis Tool. In STAT, it models penetrations as a series of state changes that lead from an initial secure state to a target compromised state. With the help of state transition diagrams, the graphical representation method can describe more complex penetration scenarios than rule-based intrusion detection systems. When performed in real-time, STAT can use audit data to track user behaviors and determine if a user's current actions represent a threat to security. STAT can perform both on-line and off-line intrusion analysis. In the off-line mode, STAT will use stored audit records to trace suspected illegal behavior. In the on-line mode, each penetration rule-chain is translated to a scenario plug-in and dynamically loaded to the STAT runtime core. As soon as the audit record is generated and formatted, they will be sent to these scenario plug-ins to perform intrusion detection analysis in real-time.

## 3. IPSW System Architecture

In order to design a better intrusion prevention engine, we first discuss the implementation issues observed in the related works and then, bring up the corresponding design strategies to overcome those problems. Based on the chosen strategies, we designed a real-time intrusion prevention system on Linux platform, which will be described in detail in section 3.2.

### 3.1 Implementation issues and strategies

There are several ways to provide security-wrapping service. One possible method is to link the application to a different library that contains wrapped functions [2]. However, every program that linked statically must be re-linked with the new modified version of library. To get rid of re-linking overhead, user-level process tracing techniques such as *ptrace* or */proc* filesystem are alternative choices [5][6]. However, there will be two context switches per system call interception that may decrease performance dramatically. In order to provide intrusion prevention with unnoticeable performance degradation, we chose kernel-level system call interception technique as our basis. Kernel-level system call interception is achieved by altering the interested system call table entry to the wrapped one. For each corresponding wrapped function, it can first perform some pre-actions, then invoke the original

kernel function and finally perform the post-actions if required. In the simplest situation, the wrapped function is able to call the original kernel function only. It can do neither the pre-actions nor the post-actions. In this case, the only extra system overheads are two jump instructions. And compared to the context switch time needed in user-level system call interception, the cost of jump instruction is negligible.

To write a rule for a rule-based IDS system, programmers have to learn the rule language for the specific IDS product. However, to write a perfect chain of rules, the programmer has to be an expert in that IDS product domain. Besides, in order to specify complicated intrusion behaviors, the rule chain will be very complex and uneasy to maintain. To lower the doorsill for usage and to make things simple, we chose to specify the intrusion scenario as a state transition diagram.

### 3.2 Architecture Overview

Based on the chosen strategies, our proposed system, IPSW(Intrusion Prevention System using Wrapper), is implemented on Linux platform. It's system architecture is shown in Figure 1. The core wrapper components are resident in Linux kernel in the form of Linux loadable kernel module and responsible for distinct jobs respectively. There are four major components in this system.

(1) **Wrapper Driver (WD)**. WD is the bridge between application-level UIs and OS-level components. The State-based rule configuration interface can issue commands to wrapper driver, and via the wrapper driver, to ask the wrapper manager to update certain system information.
(2) **Wrapper Manager (WM)**. WM is the most important component in this system. It is responsible for the registration of penetration templates, and for monitoring related tasks. To carry on the registration of penetration templates, the WM will produce a new penetration template data structure, fill in the nodes, arcs and actions received from WD, and then insert this data structure to the penetration table resident in Wrapper Information Center. When the system is running, the WM is responsible to search the penetration table to determine whether the newly created process is the suspect defined in penetration table, and then to update the status of each monitored process.
(3) **Wrapper Information Center (WIC)**. There are two major system tables in the WIC. One is the penetration template table, which is used to store all the penetration scenarios defined by users. The other is the penetration instance table, which is used to store the FSMs of all the suspected process.
(4) **State-based Rule Configuration Interface (SRCI)**. SRCI is used to specify a penetration scenario in the form of state-transition diagram.

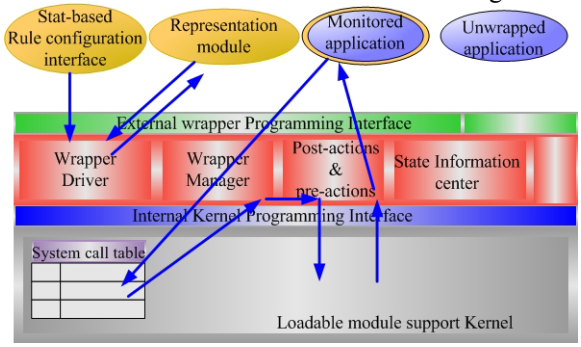


Figure 1. System Architecture of IPSW.

## 4. System Design and Implementation

As shown in Figure 2, it is the overall usage scenario of our IPSW system. In this section, we describe how to use the wrapper system to demonstrate how we design the whole system.

In the very first time, user has to do rule configuration through SRCI. After that, whenever there is a new process created (either by *fork* or *exec* system call family), the wrapper manager will check Penetration Template Table to see whether this process should be monitored. Once the process is being judged a suspected attack process, the wrapper manager will create a penetration instance and insert it into Penetration Instance Table for further tracing.

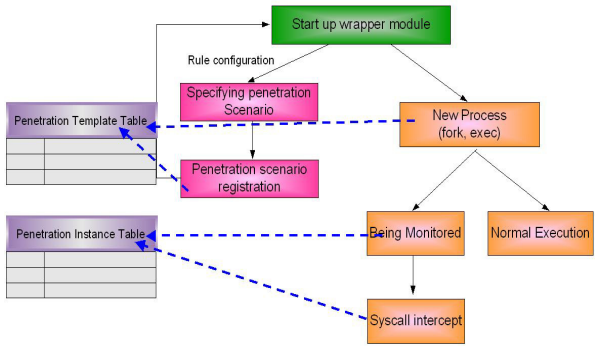


Figure 2. IPSW Overall usage scenario.

### 4.1 Rule Configuration

There are two steps to create a new rule. The first step is Penetration Scenario Specification and the second step is Penetration Scenario Registration.

To specify a penetration scenario for IPSW, user has to draw a state transition diagram like the one in Figure 3 to characterize "in what situations, the newly created process should be monitored by this template". The diagram represents a Finite State Machine (FSM). In our prototype, the prerequisite is an expression. If the evaluation of this expression is

true, then the newly created process should be monitored using this FSM. Afterwards, user can add nodes and arcs to represent the process behavior that he or she would like to trap. Each arc may be a single system call invocation, a compound event that represents a sequence of system calls, or a certain predefined penetration scenario. On each arc, programmer can write pre-actions and post-actions related to this event. These actions will be compiled as object code and will be executed dynamically when this corresponding event is triggered. In addition, the system provides a timer to trigger a certain transition and action. On the final state, user can specify what action the system should perform. In our prototype, there are three actions the system can do when facing a compromised state: (1) terminate the execution, (2) log the event in a file, and (3) wait for the decision from the user.

After creating the state transition diagram of certain penetration scenario, SRCI will parse the penetration diagram and request the wrapper driver to insert this penetration template into Penetration Template Table (PTT) for future use.

### 4.2 Process Life Cycle

Any process created by *fork* or *exec* system call families will have three stages in its life cycle: newly created, execution stage, and termination stage.

The *wrapper manager* will intercept every *fork* and *exec* system call. In this way, every newly created process will be under the control of the *wrapper manager* and has no way to escape from the security checking. Whenever there is a newly created process, the *wrapper manager* has to search the Penetration Template Table to decide which the templates should be used to monitor this process. Once the newly created process is judged as a suspected one for certain penetration scenario, the *wrapper manager* will create a penetration instance for this template, and insert these instances into Penetration Instance Table. Each process can be monitored by "stack" of templates, as long as the prerequisite of this process is satisfied with target penetration template. It means that each process can be traced concurrently among related penetration templates during one system call to another. If multiple penetration instances are waiting for the same event to make a transition, these penetration instances will turn to the next state as defined in their state transition diagram once the specific event is triggered.

Whenever there is a system call being intercepted, the execution control will transfer to the *wrapper manager*. Based on the process id, the *wrapper manager* can find out all the penetration instances related to this process. For each penetration instance waiting for this system call, the *wrapper manager* will perform its pre-action first. If the pre-action finds something illegal, it can turn on the illegal flag and the *wrapper manger* will not perform the original kernel function. If the illegal flag is not turn on, the *wrapper manager* will check whether the next state of this instance is the compromised state. If the next state were the compromised state, the *wrapper manager* would not perform the original kernel function either and instead, the wrapper manager will perform the punitive sanctions defined in the next compromised state. Afterwards, the *wrapper manager* will do the post-action we specified.

Only the process that abides by the law can execute to the end. Otherwise, it will be punished, either terminated immediately or forced to wait for the judgment from users. However, normal process without being monitored will execute as if the wrapper system does not exist.

## 5. Examples to use the IPSW

In this section, we introduce two examples to demonstrate the capability of our proposed system.

### 5.1 Deny a Socket Connection

As shown in Figure 3, a TCP network connection is created by issuing those system calls shown on arcs. In this example, any process monitored by this penetration template would be blocked if it is trying to establish a network connection. There is no need to perform any pre-actions and post-actions. When the execution reaches state s4, and there is an "accept" system call request issued by this process, the wrapper system will log the error message and terminate the process.

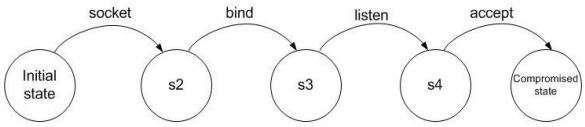


Figure 3. Deny a socket connection

### 5.2 Prevention From Virus Infection

A virus is a piece of code that copies itself into another program. Based on this definition of virus, the state transition diagram that represents the intrusion behavior of virus can be designed as shown in Figure 4.

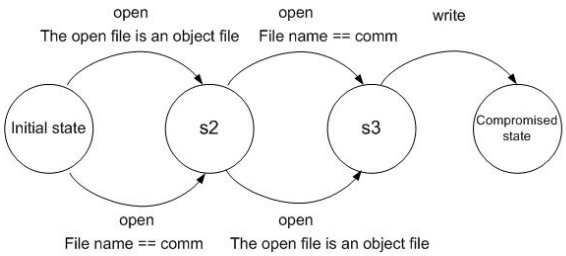


Figure 4. Virus behavior as a state diagram

The virus process can *open* an object file to be the infection target, and then *open* itself to prepare the virus body. The order can be diversified; hence we use two transitions to let either way go. When the execution reaches state s3, and there is a "*write*" system call request issued by this process that wants to write into this open object file, the wrapper system will log the event and terminate the process.

## 6. Discussions and Conclusions

There are several researches regarding intrusion detections and intrusion preventions as described in related works. All these researches have their pros and cons. In this section, we will discuss the difference between previous approaches and our own approach. Then, there will be a summary about the contributions of this paper. Finally, we will present some ideas that can be used for future investigations.

### 6.1 Strength of our approach

The most important key features of this paper are kernel-level intrusion prevention and state-transition-based rule configuration interface.

Based on the kernel-level process tracing technique, the wrapper system provides a way to monitor only suspected processes with little overhead transparently and stops the malicious process from proceeding before disaster taking place.

Based on the graphical rule configuration interface, it is more practical to specify complicated intrusion scenario in a finite state machine.

In Table 2, we compare the difference between related existing approaches and our own approach.

| | Our approach | GSW | KLW | STAT |
|---|---|---|---|---|
| Real-time intrusion prevention | YES | YES | YES | NO |
| Graphical rule configuration | YES | NO | NO | YES |
| Customized rule configuration | YES | YES | NO | YES |
| Transparency | YES | YES | YES | YES |
| Non-root usage mode | YES | NO | NO | NO |
| Partial interception | YES | NO | NO | N/A |
| Timmer support | YES | NO | NO | NO |

Table 1. Comparison between existing approaches and our approach.

### 6.2 Performance Evaluation

In order to evaluate the overhead incurred by the IPSW, we first run a small program that *opens* and *writes* a certain file in 1000, 10000 and 50000 times respectively. This evaluation program will measure the time spent both when the "Prevention From Virus Infection Wrapper (PVIW)" is loaded and not loaded, as shown in Table 2.

Each testing result is measured by "*gettimeofday*" system call, and the penalty is calculated using the following formula:

$$penalty = \frac{(TimeWithPVIWinstalled) - (TimeWithoutPVIWinstalled)}{TimeWithoutPVIWinstalled} x100\%$$

| | 1000 times | 10000 times | 50000 times |
|---|---|---|---|
| No PVIW | 75756 $\mu s$ | 726339 $\mu s$ | 3353957 $\mu s$ |
| PVIW installed | 81892 $\mu s$ | 800281 $\mu s$ | 3710352 $\mu s$ |
| Penalty | 8.10 % | 10.1% | 10.6% |

Table 2. PVIW performance evaluation of IO-bound program.

In this evaluation result, the overall performance overhead is about 10%. The PVIW wrapper will intercept the following system calls: "*execve*", "*open*", "*write*". For this reason, intensive IO operations make the whole system with PVIW loaded have 10% performance downgrade.

In order to evaluate the performance of CPU-bound program, we run another program that does 100 by 100 matrix multiplication in 1000, 10000 and 50000 times respectively. This evaluation program will also measure the time spent in both when the PVIW is loaded and not loaded. Table 3 shows the testing result. In the same manner, each testing result is measured by "*gettimeofday*" system call and the penalty is calculated using the above formula.

| | 1000 times | 10000 times | 50000 times |
|---|---|---|---|
| No PVIW | 192970 $\mu s$ | 1922763 $\mu s$ | 9620897 $\mu s$ |
| PVIW installed | 199106 $\mu s$ | 1986736 $\mu s$ | 9840510 $\mu s$ |
| Penalty | 3.18% | 3.32% | 2.28% |

Table 3. PVIW performance evaluation of CPU-bound program

In this evaluation result, the overall performance overhead is about 2% ~ 4%. In this case, the testing CPU-bound program has only computation jobs. Hence, it makes the system suffer from much lower overhead. However, during the

execution time of this CPU-bound program, the tested operating system had done several context switches for other programs, and the PVIW continually intercepted every *execve* system call to see whether the newly created process is a suspected one or not. Therefore, there is still a little overhead even though the CPU-bound testing program had no IO operations when the PVIW is installed.

### 6.3 Future Work

By dragging and drawing, users can easily specify penetration scenarios in an FSM diagram. However, user has to be familiar with various system calls and the nature of attack can he specify a good penetration diagram. With these considerations, the most important future work is to provide higher level description language other than system calls in rule configuration module, such as C-library or short human language sentence. Each human language sentence is a small pattern that matches a basic attacking action. Based on these patterns, users can build a more complicated scenario without knowing or reinventing the wheel of detail actions.

In addition, a distributed IPSW would be a better intrusion prevention system. Every IPSW agent can forward their detection result to dedicated wrapper servers as shown in Figure 5. These dedicated wrapper servers collect all information from network and analysis whether there is some kind of distributed Denial of Service (DDoS) is undertaking. Once there is someplace under attacked, the dedicated IPSW wrapper system can inform other agents to take prevention actions, such as updating their penetration rules. The distributed wrapper system will block DDoS intrusions as soon as there are some workstations reporting their auditing results. Installing the distributed wrapper system is equivalent to set up a neural network and a slight abnormal behavior will cause the whole system in action.

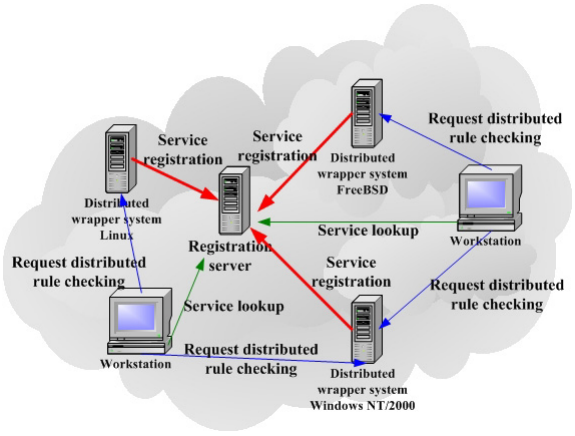


Figure 5. Distributed IPSW.

## Reference


[1] Calvin Ko, Timothy Fraser, LeeBadger, Doublas Kilpatrick, " Detecting and Countering System Intrusions Using Software Wrapper," in *Proceedings of the 9th Usenix Security Symposium*, 2000, pp.145-156.

[2] Ghosh A., Schmid M., Hill F., "Wrapping Windows NT Software For Robustness," in *Symposium on Fault-Tolerant Computing*, 1999, pp. 344-347.

[3] Giovanni Vigna and Richard A. Kemmerer. "NetSTAT: A Network-based Intrusion Detection System," *Journal of Computer Security*, Vol. 7, 1999, pp. 37-71.

[4] Giovanni Vigna, Steve T. Eckmann, Richard A. Kemmerer. "The STAT Tool Suite," in *Proceedings of DISCEX 2000*, Vol. 2, 2000, pp. 46-55.

[5] Ian Goldberg, David Wanger, Randi Thomas, "A Secure Environment for Untrusted Helper Application," in *Proceedings of the 6th Usenix Security Symposium*, 1996, pp. 1-13.

[6] K. Jain, R. Sekar, "User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement," in *Proceedings of the ISOC Network and Distributed Security Symposium*, 2000, pp. 19-34.

[7] Koral Ilgun, Richard A. Kemmerer, Phillip A. Porras, "State Transition Analysis: A Rule-Based Intrusion Detection Approach," *IEEE Transactions on Software Engineering*, 1995, pp. 181~199.

[8] Massimo Bernaschi, Emanuele Gabrielli, Luigi V. Mancini. "REMUS: A Security-Enhanced Operating System," *ACM Transactions on Information and System Security*, 2002, pp. 36-61.

[9] Mitchem, T., Lu R., O'Brien R., "Linux Ker nel Loadable Wrappers," in *Proceedings of DISCEX 2000*, Vol. 2, 2000, pp.296-307.

[10] Steven T. Eckmann, Giovanni Vigna, Richard A. Kemmerer. "STATL: An Attack Language for State-based Intrusion Detection," *Journal of Computer Security*, 2002, pp. 71~103.

[11] Timothy Fraser, LeeBadger, Mark Feldman, " Hardening COTS Software with Generic Software Wrappers," in *Proceeding of the 1999 IEEE Symposium on Security and Privacy*, 1999, pp. 2-16.

[12] Entercept™ Security Technologies. "System Call Interception". http://www.entercept.com/whitepaper/systemcalls/index.asp

[13] Pragmatic/THC. "(nearly) Complete Linux Loadable Kernel Modules". http://www.thc.org/papers/LKM_HACKING.html

[14] STAT Homepage. http://www.cs.ucsb.edu/~rsg/STAT/