Real-Time Intruder Tracing through Self-Replication

Heejin Jang and Sangwook Kim

Dept. of Computer Science, Kyungpook National University,
1370, Sankyuk-dong, Buk-gu, Daegu, Korea
{janghj, swkim}@cs.knu.ac.kr

Abstract. Since current internet intruders conceal their real identity by distributed or disguised attacks, it is not easy to deal with intruders properly only with an ex post facto chase. Therefore, it needs to trace the intruder in real time. Existing real-time intruder tracing systems has a spatial restriction. The security domain remains unchanged if there is no system security officer's intervention after installing the tracing system. It is impossible to respond to an attack which is done out of the security domain. This paper proposes self-replication mechanism, a new approach to real-time intruder tracing, minimizing a spatial limitation of traceable domain. The real-time tracing supports prompt response to the intrusion, detection of target host and laundering hosts. It also enhances the possibility of intruder identification. Collected data during the real-time tracing can be used to generate a hacking scenario database and can be used as legal evidence.

1 Introduction

An identification service is a service which identifies which person is responsible for a particular activity on a computer or network [1]. Currently, most internet attackers disguise their locations by attacking their targets indirectly via previously-compromised intermediary hosts [2,3]. They also erase their marks on previous hosts where they have passed. These techniques make it virtually impossible for the system security officer of the final target system to trace back an intruder in order to disclose intruder's identity post factum. Chasing after the intruder in real time can be an alternative. The real-time tracing supports prompt response to the intrusion, detection of target host and laundering hosts. It also enhances the possibility of intruder identification.

There are several approaches that have been developed to trace an intruder. They fall into two groups such as an ex post facto tracing facility and a real-time identification service [1]. The first type of the intruder tracing approach contains reactive tracing mechanisms. In this approach, before a problem happens, no global accounting is done. But once it happens, the activity is traced back to the origin. Caller Identification System (CIS)[4] is along this approach. It is based on the premise that each host on the network has its own tracing system. The second type, the real-time identification service, attempts to trace all individuals in a network by the user ID's. The Distributed Intrusion Detection System (DIDS)[5] developed at UC

A.H. Chan and V. Gligor (Eds.): ISC 2002, LNCS 2433, pp. 1–16, 2002.

Davis is an example of a system which did this for a single local area network. It tracks all TCP connections and all logins on the network. It maintains a notion of a Network Identifier at all times for all activities on the system. Its major disadvantage is that DIDS can account the activities only when those stay in the DIDS domain. As we have seen from above, it is possible for the existing intruder tracing systems to keep track of the intruder if they are installed in the hosts on the intrusive path in advance. That is, the biggest problem in the existing real-time intruder tracing approaches is a restriction on the traceable domain.

As a solution, this paper presents the Self-Replication mechanism that meets the aforementioned requirements. It also introduces the HUNTER which is a real-time intruder tracing system based on the Self-Replication mechanism. The Self-Replication mechanism keeps track of a new connection caused by the intruder and replicates the security scheme to the target host through the connection. It broadens the security domain dynamically following the intruder's shifting path. It means that the traceable domain is extended. The HUNTER traces an intruder and gathers information about him/her. Collected data about an intruder can be used to generate a hacking scenario database and can be used as legal evidence. If an intruder attempts to access the source host while attacking the trusted domain, the System Security Officer (SSO) could determine the origin of the attack. The Self-Replication mechanism is applicable to general security solutions, such as system/network level monitoring systems, intrusion detection system, intruder tracing system and intrusion response system.

The remainder of this paper is structured as follows. Section 2 defines terminology. Section 3 proposes the Self-Replication mechanism for real-time intruder tracing. Section 4 shows that the security domain for real-time tracing is extended through the Self-Replication. Section 5 presents the architecture, the working model of the HUNTER and an implementation example. Section 6 shows performance evaluation. Finally, section 7 draws some conclusions and outlines directions for future research.

2 Preliminaries

We first define terminology.

2.1 States, Events, and Logs

We assume that the set of entities O and a set of well-formed commands E can characterize the computer system completely [6]. O is what the system is composed of and E is the set of events that can cause it to change. Following [7], a system state s is a 1-tuple (O). The collection S of all possible states is the state space. The relevant part of the system state $\sigma \subseteq s$ is the subset of (O). The collection Σ of the relevant parts of all possible system states is the relevant state space.

Monitoring activity is indispensable for security management. Monitoring is classified into two types, system state monitoring and change monitoring [7]. System state monitoring periodically records the relevant components of the state of the system. Change monitoring records the specific event or action that causes altering relevant component of the state of the system as well as the new values of those

components. The output of each monitoring activity is a log $L=\{m_0, m_1, \ldots, m_p\}$, $m_k \in I$ for all $k \geq 0$. Monitoring of relevant state space Σ makes the state log entry $I = N_O \times N_V$ and that of relevant state space Σ and event E generate the change log entry $I = N_S \times N_O \times N_V \times N_E$. N_S are the names of users who cause events, N_O the names of the objects such as files or devices, N_V the new values of the objects, and N_E the names of the events.

2.2 A Trusted Domain and a Security Domain

We here define a trusted domain. The trusted domain D_t is composed of several domains including single administrative domains or cooperative domains. In the trusted domain, each administrator of constituent domains also has the administrative privilege in other domains.

If we consider that a security scheme is a way of controlling the security of systems or networks, the security domain [8] D_S is the set of machines and networks which have the same security scheme. Each of the heterogeneous security management systems generates its own security domain. Single administrative domain includes more than one security domain and single security domain is made up of more than one host. The security domain D_S has a static characteristic, for it does not change if the SSO does not install a security management system additionally. Those attributes of a security domain cause spatial restriction for general security management. Especially, it is the point at issue to identify intruders. If T is a security scheme, a security domain D_S controlled by T is defined by the function *dom(T)*. The result of *dom(T)* is composed of various data representing the domain, such as network topology information N_t, network component information N_c and monitoring information M which is basically obtained by T. N_c contains information about hardware, operating system, services to be provided and etc. M is a set of log which is defined above. Since N_t and N_c have a regular effect on the extension of the security domain as expected, we consider only M that decides attributes of *dom(T)*. M consists of $m_1, m_2, m_3, \ldots, m_k, \ldots$ in which m_i is single monitoring information, i.e. a log entry. Each m_i has an occurrence time, denoted by $t(m_i)$. They are totally ordered, that is, $t(m_i) \leq t(m_{i+1})$ for all $t \geq 1$. M has spatial location as well as temporal sequence.

2.3 Real-Time Intruder Tracing

When a user on a host H_0 logs into another host H_1 via a network, a TCP connection C_1 is established between them. The connection object C_j ($j \geq 0$) is constructed as <*connectionType, fromHostID, toHostID, fromUserID, toUserID, toPasswd, Time*>. *connectionType* is the type of connection and *fromHostID* and *fromUserID* are the source host id and the user id on the source. *toHostID* is the target host id, *toUserID* and *toPasswd* are the user id and the password information on the target and *Time* indicates when the connection occurs.

When the user logs from H_1 into another host H_2, and then $H_3, \ldots, H_n$ successively in the same way, TCP connections $C_2, C_3, \ldots, C_n$ are established respectively on each

link between the computers. We refer to this sequence of connections $CC=<C_1, C_2, \ldots, C_n>$ as an extended connection, or a connection chain [9].

The task of a real-time intruder tracing is to provide intruder's movement path completely from source host to target host. In order to do this, the security domain must be secured.

3 Self-Replication

The Self-Replication mechanism supports dynamic extension of the security domain. It observes behavior of the user who is presumed as an intruder and acquires activity and identity information of the user. Using these data, it replicates itself or any other security scheme automatically into the hosts where an intruder has passed. Consequently, it broadens the security domain for data collection used for security management and intruder tracing. The Self-Replication mechanism could not only work independently but also operate together with any security scheme. The Self-Replication mechanism consists of monitoring and filtering, replication [10] and self-protection. Monitoring in the Self-Replication mechanism is a data-collecting phase for replication. It is done for specific users or all users who enter the trusted domain. The output of each monitoring activity is a log L. For replication, the Self-Replication mechanism filters some useful states or events among logs, which is related to establishing of new connections. Interesting states are aspects of important objects which can affect system security. They include states of *setuid* and *setgid* files, users with superuser privilege, or integrity of important files. Event under the close observation is the generation of new connections caused by user session creation event, account change event or intrusive behavior event. The user session creation event contains commands such as *telnet*, *ftp*, *rlogin*, *rexec* and *rsh*. The account change event includes gain of other user's privilege using *su* command. The intrusive behavior event comprises illegal acquisition of other user's or superuser privilege by buffer overflow attack, backdoor, and creation of malicious process or new trap.

After filtering, a point of time and a target host for replication have to be chosen. The Self-Replication mechanism decides the target host and starts to replicate itself to the host when an intruder succeeds in connecting with another host. With respect to Unix or Linux, there are various methods to connect two hosts [11]. We just take the connection through the medium object into consideration in this paper. The Self-Replication mechanism provides all users with medium objects which work normally but are controllable by this mechanism. It delivers modules and events for replication to the target host via the medium object, especially the medium object with a command processing function such as a pseudo terminal with a working shell.

Fig. 1 illustrates the event transmission through the pseudo terminal object. A lower hexahedron shows the Self-Replication mechanism running on the host. An upper hexahedron denoted as US_X is a user space in each host. It includes all actions by a specific user and every resource related to those actions. The front rectangle of the US_X is a perceptible part to the user such as standard input or standard output. As it goes back, it gets closer to the operating system. A solid arrow shows transfer of specific event $e_r \in E_R$ where E_R is a subset of E and a set of events such as copying, compiling or execution command for replication. A dotted arrow indicates forwarding

of normal events $e_n \in (E - E_R)$. For example, a user U_R in a host H_R which already has the Self-Replication mechanism *SM* attacks a host H_T and U_R becomes U_T who has superuser privilege in H_T. A connection C_n is set up between two pseudo terminal objects, $MO_{U_R}^{C_n}$ which is allocated for U_R in the H_R by *SM* and $MO_{U_T}^{C_n}$ for U_T in the H_T by an operating system. Therefore, it is possible to send an event to $MO_{U_T}^{C_n}$ via $MO_{U_R}^{C_n}$ in order to remotely execute any command in H_T from H_R. A normal event e_n is delivered to H_T via $MO_{U_R}^{C_n}$ and $MO_{U_T}^{C_n}$ so that the user U_R can accomplish the event e_n with the privilege of U_T in the H_T. The event e_n is carried out in the H_T normally and the result e_n' is showed to the user U_R. For example, when a command *telnet* from H_R to H_T succeeds, a pseudo terminal is allocated [11]. Then if *ls* command is transmitted through a pseudo terminal of H_R, it is executed at H_T and the result is showed at the pseudo terminal of H_R. The *SM* makes the replication event e_r which is delivered to $MO_{U_T}^{C_n}$ via $MO_{U_R}^{C_n}$ and executed by means of superuser authority in the H_T. The result at H_T comes back to US_R but is not showed to U_R on $MO_{U_R}^{C_n}$ to keep any intruder from watching the whole process. It protects the replication process and *SM* itself from detection by U_R. As a result, *SM* of the host H_R replicates itself to the host H_T and duplicated Self-Replication mechanism *SM'* operates at H_T.

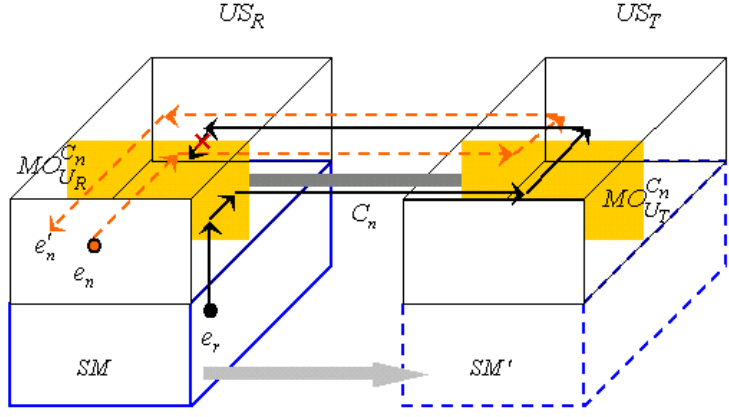

Fig. 1. Event Transmission through the Pseudo Terminal Object

Fig. 2A and 2B depict the replication protocols and their timing diagrams which are performed when the intruder succeeds in penetrating the target host H_T from the host H_R through the pseudo terminal object. RPC_X shows the replication status in each host. When a connection is established by the specific event *e* in the state of RPC_P, host H_R sends an *authreq* message to request the authentication for replication. If there is the same security scheme or faked scheme, H_T delivers a response message *authres* like in the Fig. 2A. H_R certifies legitimacy of the scheme and terminates the replication process. If the host H_T cannot receive *authres* during a lapse of a specific

time, the host H_R enters a replication ready state of RPC_R and sends a *readyreq* message to check the intruder's environment in the H_T in the Fig. 2B. The target host enters the state RPC_R and transfers a *readyres* message which is the information about the intruder's execution environment in the target. After recognizing the intruder's environment, H_R enters a replication execution state of RPC_M and transmits modules and events for replication with *rpcout* message to H_T. H_T in the state of RPC_M executes commands from the host H_R. The Self-Replication mechanism is set up in the host H_T and starts inspecting the host and the specified intruder. And chasing an intruder continues. H_T sends *termreq* message to inform H_R of completion of the replication process. Then H_R enters a replication completion mode RPC_D and puts H_T into the state RPC_D by transmitting the *termres* message. Since the replication process is hidden from an intruder and the intruder's execution environment is maintained in the target host, the intruder cannot recognize the process. By using two self-protection methods, track erasing and camouflaging (explained in section 5.1), it protects the Self-Replication mechanism. Timing diagrams of Fig. 2A and 2B show the temporal relation among replication states of H_R, input to H_R and output from H_R.

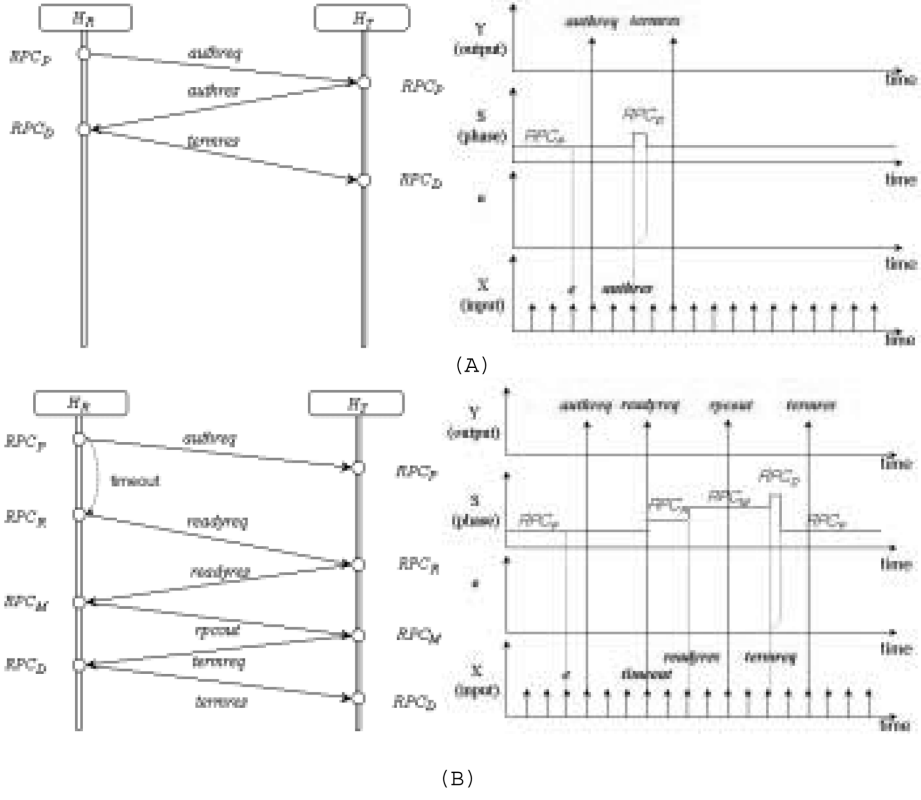

Fig. 2. (A) Replication Protocol and Timing Diagram in case there is Self-Replication mechanism in the target host (B) Replication Protocol and Timing Diagram in case there is no known Self-Replication mechanism in the target host

4 Security Domain Extension for Real-Time Intruder Tracing

In this section, we show that the security domain expands dynamically by the Self-Replication mechanism. If *SS* is a security scheme based on the Self-Replication mechanism, a security domain D_S controlled by *SS* is defined by the function *dom(SS)*. We consider only *M* that decides attributes of *dom(SS)*(see Section 2.2). Given two sets of monitoring information M^{H_1} and M^{H_2} which are gathered in hosts H_1 and H_2, the merge of these two sets is partial information of *M*, denoted by $M^{H_1} \oplus M^{H_2}$. If we assume that login id of the user *A* is same in every host in the trusted domain, M_A, a set of monitoring information about the activities of a user *A* in only two consecutive hosts is $M_A = M_A^{H_1} \oplus M_A^{H_2}$ where $M_A^{H_1}$ is a set of monitoring information about user *A* in host H_1. M_A is $M_A^{H_1} \oplus M_A^{H_2}$ = $m_{H_1 1}, m_{H_1 2}, ..., m_{H_1 k}, m_{H_2 1}, m_{H_2 2}, ..., m_{H_2 p}$ = $m_1, m_2, ..., m_{k+p}$ if and only if there exist two sequences $H_1 1, H_1 2, ..., H_1 k$ and $H_2 1, H_2 2, ..., H_2 p$ of the sequence 1, 2, … , *k*+*p* s.t. $M_A^{H_1} = \{ m_{H_1 1}, m_{H_1 2}, ..., m_{H_1 k} \}$ and $M_A^{H_2} = \{ m_{H_2 1}, m_{H_2 2}, ..., m_{H_2 p} \}$(see Section 2.2). The Self-Replication mechanism can recognize the changes of user's identity and monitor all behaviors of him/her while the user travels the trusted network. Therefore, if the user *A* passes through hosts H_1, H_2, … , H_n ($n \geq 2$) sequentially and produces logs such as $M_A^{H_1}$, $M_A^{H_2}$, … , $M_A^{H_n}$ in each host, the resulting set of monitoring information about user *A* can be extended to $M_A = M_A^{H_1} \oplus M_A^{H_2} \oplus \ldots \oplus M_A^{H_n}$. If an intruder begins to attack the trusted domain by penetrating the host which has a security scheme with the Self-Replication mechanism *SS* uniquely in the trusted domain, the result of security management by the *SS* is equal to that by installing and executing the scheme in every host on the intrusive path in the trusted domain. The specific user's sequence of behaviors in every host on the intrusive path is equal to the union of monitoring information sets each of which is gathered about a user in each host on the path by the Self-Replication mechanism.

Fig. 3 illustrates the security domain extension using the Self-Replication mechanism in a part of the trusted domain. Initially, there is only one host H_X with the *SS* in the trusted domain. We assume that an intruder passes through the host H_X first to break into the trusted domain from outside and continues to attack H_Y via H_X and then H_W via H_Y. Early security domain D_S controlled by *SS* is *dom(SS)*, denoted as A in the Fig. 3. When an intruder succeeds to attack via path2, the *SS* in host H_X replicates itself to H_Y. Let SS_{rH_Y} be the replicated security scheme in H_Y, D_S is expanded to *dom(SS)* $\oplus$ *dom*(SS_{rH_Y}) (denoted as B). The D_S is enlarged to *dom(SS)* $\oplus$ *dom*(SS_{rH_Y}) $\oplus$ *dom*(SS_{rH_W}) (denoted as C) by the attack via path 3.

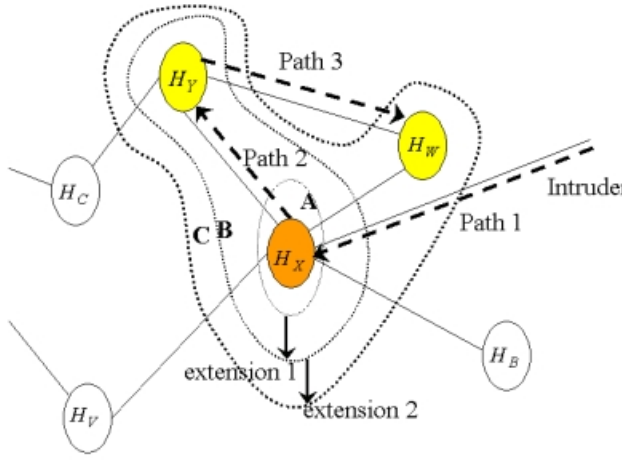

Fig. 3. Security Domain Extension for Real-Time intruder Tracing

5 Implementation

The real-time intruder tracing system, HUNTER, aims at keeping track of an intruder and, if possible, revealing the intruder's original source address and identity. Since this system is developed on the basis of the Self-Replication Mechanism, it is possible to enlarge the traceable domain following the intruder's shifting path even though a security scheme for identification is not installed in advance in all hosts within the trusted domain, unlike existing intruder tracing systems.

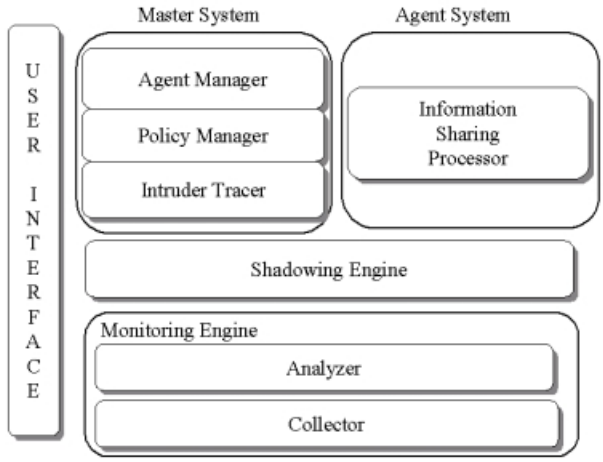

Fig. 4. System Architecture

5.1 HUNTER: Real-Time Intruder Tracing System

The HUNTER is composed of a single master system and several agent systems. This is initialized by installing a master system in the host which is the unique entrance to the trusted domain (for example, routers). Initially, there is a master system only in the trusted domain. If an intruder moves into another host via master system in the trusted domain, agent system is automatically placed into the target host through the self-replication process. As the self-replication process goes on following the intruder's movement path, the number of agent systems increase dynamically. This system is implemented in the GNU C/C++ 2.7.x.x for core modules and Java 2 SDK for the user interface on Linux 2.4.6 and Solaris 2.8. It uses MySQL 3.22.x as DBMS to store the monitoring information and JCE (Java Cryptography Enhancement) 1.2.x package for authentication and encryption between systems.

Fig. 4 describes the architecture of HUNTER. The master system and agent systems share a Monitoring Engine and a Shadowing Engine. The Monitoring Engine consists of a Collector and an Analyzer. The Collector of master system gathers activities of all users who logged in the master system. The agent system observes the user who is thought to be an intruder by any intrusion detection module. The Analyzer of master and agent systems examines each collected activity and produces the formalized object *FO*. Certain critical *FOs* are always transmitted to the master system in real-time; others are processed locally by the agent system and only summary reports are sent to the master system. A Shadowing Engine replicates the security scheme following intruder's migration. The master system manages the predefined rules to trace an intruder. The Intruder Tracer of master system extracts the useful pieces among *FOs* and constructs a connection chain which will be explained in subsequent sections. The Agent Manager controls all the distributed agent systems in the trusted domain.

The Shadowing Engine replicates the security scheme to the host on the intrusive path and supports domain extension to trace an intrusion. Fig. 5 presents the structure of the Shadowing Engine. The engine is composed of the replication module and the self-protection module.

A *FO* Filter extracts useful pieces among data sent from the Monitoring Engine and a *FO* Analyzer decides the point of time and the target host for replication. When any *FO* related to the connection is detected, *FO* Analyzer determines the target host and begins to transfer the security scheme. The Connection Manager attempts to establish the TCP connection with the target host. The Self Replication Module Manager checks the existence of the same security scheme in the target host. If there is same security scheme, the Self Replication Module Manager verifies that the installed scheme is the legal one through authentication and terminates the replication into the target host. Otherwise it lets the Remote Protocol Manager and the Remote Shell Manager send the security scheme to be copied and commands for installation, compiling and running of the duplicated modules to the target host. If above process is successful, the security scheme is set up in the target host. Since the security scheme is replicated using the pseudo terminal as a medium object, it is necessary to maintain an intruder's environment so that the intruder cannot recognize the

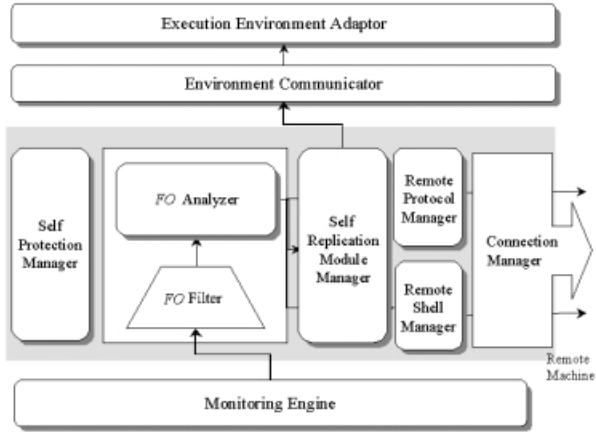

Fig. 5. Replication Engine

replication. The Environment Communicator and Execution Environment Adaptor support this maintenance. Replication protocol in the Self-Replication mechanism works as explained in section 3. Self-protection in the Self-Replication mechanism is to protect the monitoring activity itself. This plays an important role in earning some time to observe an intruder. The Self-Protection Manager in Fig. 5 attempts both to erase shadowing tracks and to blend into the normal Unix/Linux environment using camouflage. The Self-Replication mechanism carries out a number of functions to cover its trail. It erases its argument list after processing the arguments, so that the process status command would not reveal how it is invoked. It also deletes the executing binary, which would leave the data intact but unnamed, and only referenced by the execution of the Self-Replication mechanism. It uses resource limit functions to prevent a core dump. Thus, it prevents any bugs in the program from leaving telltale traces behind. In addition to erasing the tracks, camouflage is used to hide the shadowing. It is compiled under the name sh, the same name used by the Bourne Shell, a command interpreter which is often used in shell scripts and automatic commands. Even a diligent system manager would probably not notice a large number of shells running for short periods of time. Like this, it shields itself by replacing an original application program with a modified one. It sends the fake program along with the modules for replication like the trojan horse program. It can conceal processes using *ps*, *top* or *pidof* and hide files using *find*, *ls* or *du*.

5.2 Intruder Tracing by the HUNTER

This system assigns trace-id(TID) to a new user who is decided to be the intruder by any intrusion detection module and maintains a connection chain about TID. The connection chain chases the intruder's movement. The connection includes all sorts of connections which can occur through the pseudo terminal.

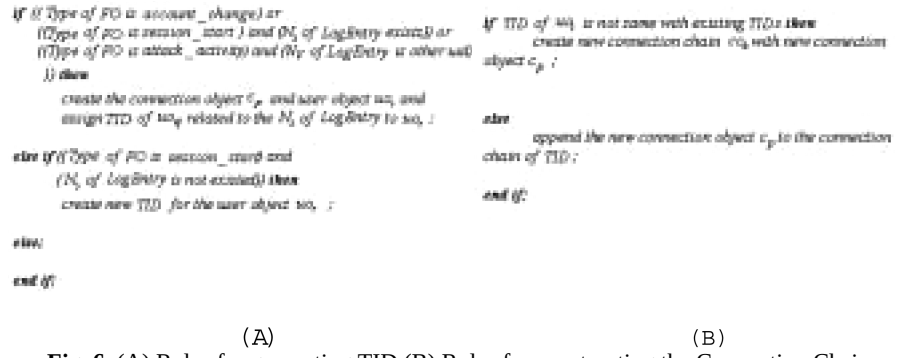

Fig. 6. (A) Rules for generating TID (B) Rules for constructing the Connection Chain

The master system monitors all users' activities on the host which the master system runs. The agent system just watches the users thought to be intruders by the master system. The targets of monitoring include attempts to connect, file access operation, process execution operation and etc. Monitoring activity records a log from which the formalized object *FO* is generated. *FO* is composed of the log collected at each agent system or the master system, the ID of the host that made the log and the *Type* field. In case of change log, *Type* may have such values as *session_start* for user session creation event, *account_change* for account change, *attack_activity* for intrusive behavior event and etc. A useful data abstracted from *FO* contains a connection object and a user object. Concerning Unix and Linux, the three ways to create a new connection are for a user to login from a terminal, console, or off-LAN source, to login locally or remotely from an existing user object legitimately, and to gain other user's privilege by illegal method such as a buffer overflow attack. These connections make new user objects and connection objects. The master system receives those objects from agent systems. It constructs a connection chain from connection objects and tries to associate the user object with an existing TID or allow the user object a new TID. We consider a user object $uo_i \in UO$ ($i \geq 0$) to be the 4-tuple <*TID, UserID, HostID, Time*> where *UO* is a set of user objects in the trusted domain. After TID generating rule is applied to the user object, value for TID is assigned.

TID and a connection chain play important parts in tracing an intruder. TID provides a unique identifier for the user who continues to attack across several hosts. Whenever a new connection object is created, new user object is formed and applicable TID is assigned to the user object. Finding an applicable TID consists of several steps. If a user changes identity on a host, the new user object is assigned the same TID as the previous one. If a user establishes a new connection with another host, the new user object gets the same TID as that of the source user object. The new user object is assigned the same TID as the previous identity in the case where the intruder obtains the superuser privilege in the remote host using vulnerabilities of the remote server. Since the user who logins from a terminal, console, or off-LAN source does not have the previous identity, new TID is assigned to the user object. Fig. 6A describes a rule with which connection objects and user objects are generated from *FO* and TID is assigned. Each TID maintains its own single connection chain to keep track of the intruder. Whenever a user with same TID sets a new connection, the

generated connection object is appended to the established connection chain. New connection chain is created if new TID is allocated for the user object. A single connection chain $cc_i \in CC$ ($i \geq 0$) is the information which is maintained for the user whose TID is i. The connection chain is a sequence of more than one connection object. The rule for constructing the connection chain is shown in the Fig. 6B. The connection chain makes it possible to trace an intruder and disclose the source of attack and the intruder's identity.

5.3 Implementation Example

Fig. 7 presents the web-based user interface of HUNTER. Each of the upper frames of two big windows displays the connection chain for a specific TID. The two right bottom windows show the information of each connection object in the connection chain and intruder's activities in real time. The bottom frame of the left big window presents information about the source host and the user who begins the connection chain.

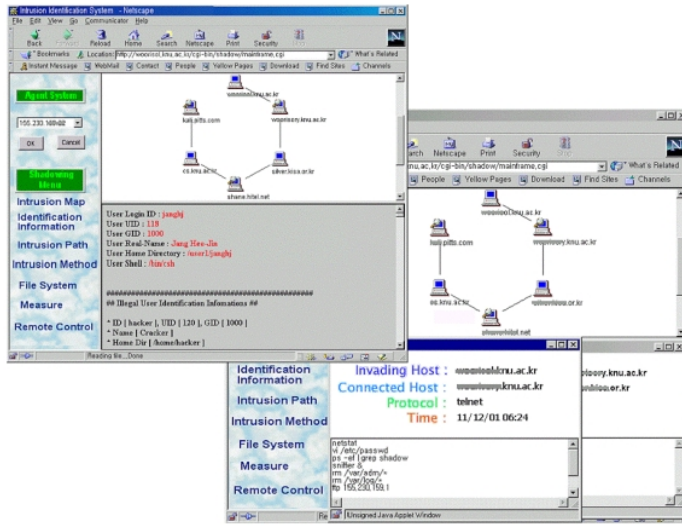

Fig. 7. User Interface (Host names are omitted for anonymity)

6 Performance Evaluations

We measured an intruder tracing rate which changes by the range of attack or intrusive path about the extent or location of the initial security domain in this experiment. An Intruder Tracing Rate Per Individual $ITRPI_i$ is a degree of tracing the intrusive path of the user i who establishes new connections within the trusted domain D_t. It is given by

$$ITRPI_i = \frac{ICC_i}{BI_i} \tag{1}$$

where BI_i is the number of connection objects generated by the user i in the D_t and ICC_i is the number of connection objects in the connection chain maintained for the user i uniquely by the HUNTER. $ITRPI_i$ has a value between 0 and 1. The value 1 of $ITRPI_i$ means that we can keep track of the specific user i completely in the trusted domain. *ITR* is the mean intruder tracing rate for all users who are inferred to be intruders in the trusted domain. It is given by

$$ITR = \frac{\sum_{i=1}^{n} ITRPI_i}{n} \tag{2}$$

where n is the number of distinctive intruders in the domain D_t.

The target network was a class C, composed of four subnets which included 48 hosts and based on the Ethernet. It was in a single trusted domain. In this experiment, we confined network components to routers or gateways, PCs and workstations, the operating system running on each host to Solaris 2.6, Red Hat Linux 6.1 or above, and services to *telnet*, *ftp*, *www* and e-mail service. In order to lower a complexity, we assumed that an SSH secure shell 3.0.0 remote root exploit vulnerability [12] was implicit in every target and intermediary host of the attack in the trusted domain. We also presumed that the only attack used SSH secure shell vulnerability and the success rate of the attack was 100%.

6.1 Intruder Tracing Rate by the Initial Location of a Security Domain

We assessed the intruder tracing rate as the location of an initial security domain changes. We assumed an initial security domain covering only one host and a specific intrusive path within the trusted domain.

Fig. 8 shows conditions for this experiment. The path included 12 hosts which were distributed in four subnets. There was only one host X with the master system in the trusted domain. In case A, X was out of the intrusive path. In case B, X was the fifth host on the intrusive path. In case C, the intruder attacked the host X first to penetrate the trusted domain. For each case, we generated 50 different intrusive paths which satisfied the above condition. 10 different users passed through different paths.

Fig. 9 shows the result of the experiment. The x-axis presents the degree of an attack advance through the intrusive path. The *ITR* indicated by y-axis is 1 if every connection caused by intruders is noticed within the trusted domain. In case A, the attacks have been advanced out of the security domain. That's why the *ITR* is 0. In case B, not all intrusion paths could be traced. It was possible to trace the path from the point of time when the intruder has gone through the master system. In case C, when the intruder has passed through the host X first to penetrate the trusted

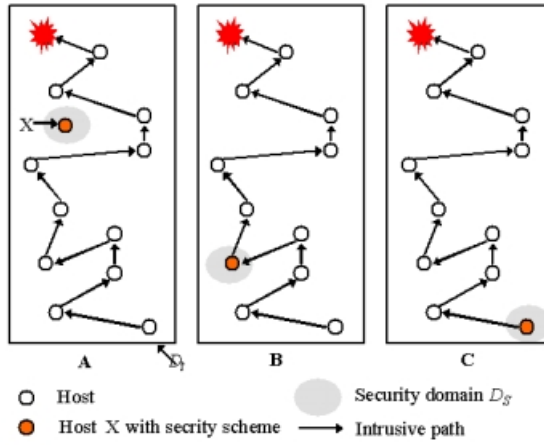

Fig. 8. Conditions for the Experiment

domain, the security domain could be extended to cover the total intrusive path within the trusted domain, making it possible to trace the intruder. This experimental result shows that the effect of the Self-Replication mechanism can be maximized if the master system is in the host which is a unique entrance to the trusted domain.

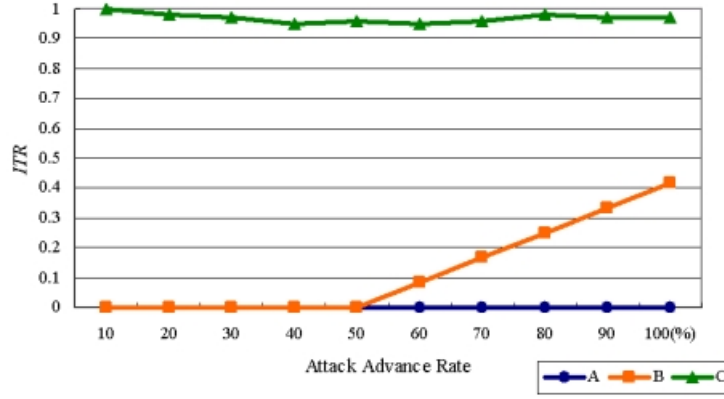

Fig. 9. Intruder Tracing Rate by the Attack Advance Rate

6.2 Intruder Tracing Rate by the Attack Range

As the attack range became wider within the trusted domain, we tested the intruder tracing rate on the condition listed in Table 1. We regarded the entire trusted domain as 100%. For the experiment, the attack range varied from 0% to 100% irrelevant to the intrusive path. However, the first penetrated host in the trusted domain was fixed and any security scheme was placed in that host in each case.

Table 1. Conditions for Evaluation

Condition / Case	The number of system with security solution on the intrusive path in advance	Installed Security Solutions
A	1(2.08%)	HUNTER without Self-Replication
B	12(25%)	HUNTER without Self-Replication
C	48(100%)	HUNTER without Self-Replication
D	1(2.08%)	HUNTER

In case of D in Table 1, we installed HUNTER into the only one host in the trusted domain. In other cases, HUNTER without the Self-Replication mechanism which cannot broaden the security domain was set up in more than one host.

Fig. 10 shows the result of experiment. In case of A, there was only one host with security solution. As an intruder extended the attack range, the *ITR* dropped rapidly. About 25% of hosts including the first attacked host in the trusted domain had the security solution in the case of B. It shows that *ITR* of case B was better than that of case A but the rate still went down as the attack has advanced along the intrusive path. In case of C, we had to install the security solution into all hosts in the trusted domain in advance. It was possible to respond to the intrusion in every host in cases of C and D. However, Case D had considerable merits over case C with respect to the cost. That's because installing and executing the security solution were performed automatically through the intrusive path in case D.

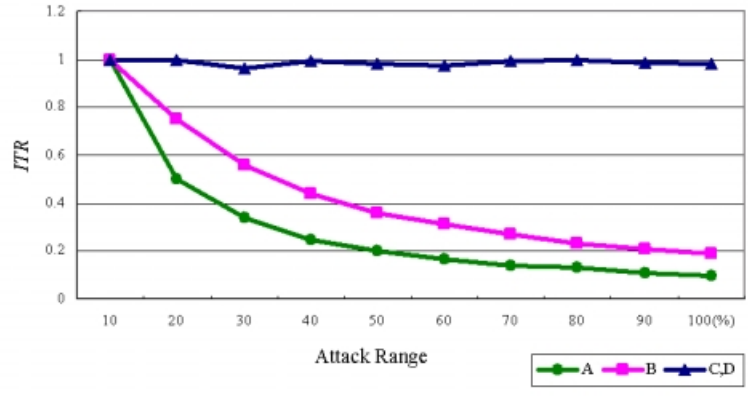

Fig. 10. The Intruder Tracing Rate by the attack range

7 Conclusions

Existing security management systems including intruder tracing systems fix their security domain after being installed in some hosts by SSOs. Therefore, it is impossible to respond to the attack properly as an intruder continues to attack across several hosts.

For this reason, this paper proposed the Self-Replication Mechanism and HUNTER which is a real-time intruder tracing system based on the mechanism. The Self-Replication Mechanism applies to the case that an intruder uses the medium object such as a pseudo terminal at least once during an attack on the trusted domain. The Self-Replication mechanism is applicable to general security solutions. The HUNTER traces an intruder and gathers information about him/her. If an intruder attempts to access the source host while attacking the trusted domain, the SSO could determine the origin of the attack. This system overcomes the restriction on the security domain under certain assumptions. Since the proposed approach in this paper traces the user who is assumed to be the intruder by any intrusion detection system, it is necessary to consult any intrusion detection system. A proper response to the attack is carried out during shadowing of the intruder.

References

1. S.S. Chen & L.T. Heberlein: Holding Intruders Accountable on the Internet. In Proceedings of the IEEE Symposium on Security and Privacy, (1995) 39–49
2. G. Eschelbeck: Active Security-A proactive approach for computer security systems. Journal of Network and Computer Applications, 23, (2000) 109–130
3. D. Schnackenberg, K. Djahandari & D. Sterne: Infrastructure for Intrusion Detection and Response, Advanced Security Research Journal, 3, (2001) 17–26
4. H.T. Jung *et al*.: Caller Identification System in the Internet Environment, In Proceedings of Usenix Security Symposium, (1993)
5. S. Snapp *et al*.: DIDS(Distributed Intrusion Detection System) – Motivation, Architecture, and an early prototype. In Proceedings of National Computer Security Conference, (1991) 167–176
6. M.R. Cornwell: A Software Engineering Approach to Designing Trustworthy Software. In Proceedings of the Symposium on Security and Privacy, (1989) 148–156
7. M. Bishop: A Model of Security Monitoring. In Proceedings of the Annual Computer Security Applications Conference, (1989) 46–52
8. S. S. Chen: Distributed tracing of intruder, Thesis of master's degree, Dept. of Computer Science, U.C.Davis. (1997)
9. K. Yoda and H. Etoh: Finding a Connection Chain for Tracing Intruders. In Proceedings of 6th European Symposium on Research in Computer Security - ESORICS 2000 LNCS - 1985, Toulouse France (2000)
10. H. Jang & S. Kim: A Self-Extension Monitoring for Security Management. In Proceeding of the 16th Annual Computer Security Applications Conference, (2000) 196–203
11. W.R. Stevens: Advanced Programming in the UNIX Environment, Addison-Wesley Publishing Company, (1992) 631–658
12. SSH Secure Shell 3.0.0 Security Advisory 2001. Found at URL: http://www.ciac.org/ciac/bulletins/l-121.shtml, CIAC, U.S. Department of Energy