Falsification Prevention and Protection Technologies and Products

A Behavior-Based Process Confinement Method and Its Application to a Server Security Solution "StarDefence"

By Masayuki NAKAE,* Ryuichi OGAWA,* Yasushi SATO† and Sonomi SHIOZAWA†

ABSTRACT Conventional server applications such as Web applications usually perform their tasks through the collaboration of several processes including CGI processes and shell processes, etc. If any of the processes are taken over by attackers, the security of the entire applications could be compromised. To protect the server applications, it is necessary to protect all related processes. We propose a behavior-based process confinement method that restricts irregular process behavior. This method prevents the process behavior from violating given rules, called Context-Sensitive Policies (CSP). CSP's specify not only a set of system calls that each process is permitted to invoke, but also the relationship between system call sequences and application-dependent specifications, so that they can correctly describe the normal behavior of server applications. This paper describes a CSP matching algorithm with actual process events and how the algorithm efficiently prevents the processes from being taken over by attacks such as code injection. This paper also describes the implementation of this method and the result of its evaluation.

KEYWORDS **System security, Process confinement, Attack prevention, Server applications, Security solution**

1. INTRODUCTION

Many companies and organizations run network services such as advertising services for their customers via the Internet and information services to their employees via their intranet. There are, however, various threats to these network services. For instance, Internet worms can quickly spread over many servers, intruders may steal confidential and proprietary information, and so on.

To protect potentially vulnerable server applications, pattern-based attack prevention methods have been proposed[1,2]. These methods detect the characteristic pattern of process behavior observed when the server application is infected by malicious codes. These methods monitor machine instruction flows, and detect illegal operations on memory such as stack overflows.

Process confinement methods have also been proposed[3-5]. These methods restrict the behavior of individual processes based on given rules, called policies, and seek to guarantee that no process will access any resources that are not authorized in the process specifications even when malicious codes are injected. For example, once a policy that prohibits a process writing the file "/etc/passwd" is issued, then the process cannot modify "/etc/passwd," even if the process is infected by malicious codes.

Since normal behavior of applications is usually application-specific, we need to enforce policies that can reflect such application dependency. The problem of existing process confinement methods is that they do not accept application-dependent policies. For example, they do not accept a policy that permits only administrators on an authorized site accessing a remote application maintenance service.

In this paper, we propose a behavior-based confinement method using policies that describe the relationship between process behavior and application-dependent specifications. We call the policy a Context-Sensitive Policy (CSP).

We also propose a server behavior model, called Behavior-Tree, to develop an efficient algorithm to compare process behavior and CSP's. The model represents the relationship between each event and the application status. Based on this method, we have developed an experimental sandbox mechanism, called S-Tracer, which can efficiently monitor each process and enforce CSP's for the processes.

In the following sections, we first describe the behavior-based process confinement method (Section

*Internet Systems Research Laboratories
†System Platform Software Development Division

2). Next we show a behavior-tree construction algorithm and a CSP matching algorithm based on the behavior-tree (Section 3). We also explore the implementation issues of S-Tracer, and its application to a practical security solution, named "StarDefence" (Section 4). Furthermore we show the evaluation results of the effectiveness, performance and administrative workload (Section 5). Finally we conclude in Section 6.

2. BEHAVIOR-BASED PROCESS CONFINEMENT

2.1 Requirements

Let us consider a Web application that has two services, Service A and Service B, as shown in **Fig. 1**. Service A provides a confidential data search service for authorized users. Service B provides a browsing service using PHP for the Internet.

If the CGI process in Service A and the PHP process in Service B are vulnerable, an attacker may access the confidential data by taking over either of the vulnerable processes and injecting malicious code into the shell process. In order to detect such malicious behavior, it is necessary to monitor not only the server processes, but also other processes of the application.

In addition, consider when Service A has an application-specific rule that it accepts only authorized clients. Then an attacker on an unauthorized client may inject malicious code into the CGI process or the shell process. If we can identify the client accessing the shell process, we can detect the violation of the application-specific rule and prevent the shell process from being taken over.

Furthermore consider when Service A has another application-specific rule that it does not accept any request of Service B. Then an attacker on an attacking client who intruded into Service B may attempt a secondary attack against the shell process via the PHP process. If we can confirm that the parent process of the shell process does not originate from Service A, then we can detect the violation of the application-specific rule and protect the shell process.

These examples show the following requirements of the process method:

(a) Comprehensiveness: Any process behavior on the server must be watched.
(b) Identifiability: Each access request (i.e. the client) that triggers each process must be identified.
(c) Traceability: The history of process execution must be traceable.
(d) Knowledge of normal behavior: The current process behavior must be validated based on the knowledge of normal process behavior including application-dependent specifications.

2.2 Proposed Method

To meet the above requirements, we construct two methods: a behavior model of the application, and a knowledge description model of the normal process behavior.

2.2.1 Behavior Model

For requirements (a), (b), and (c), we develop a behavior model, called Behavior-Tree, which is comprised of system call events of a server application.

A behavior-tree is an ordered tree comprised of process nodes and event nodes. Process nodes

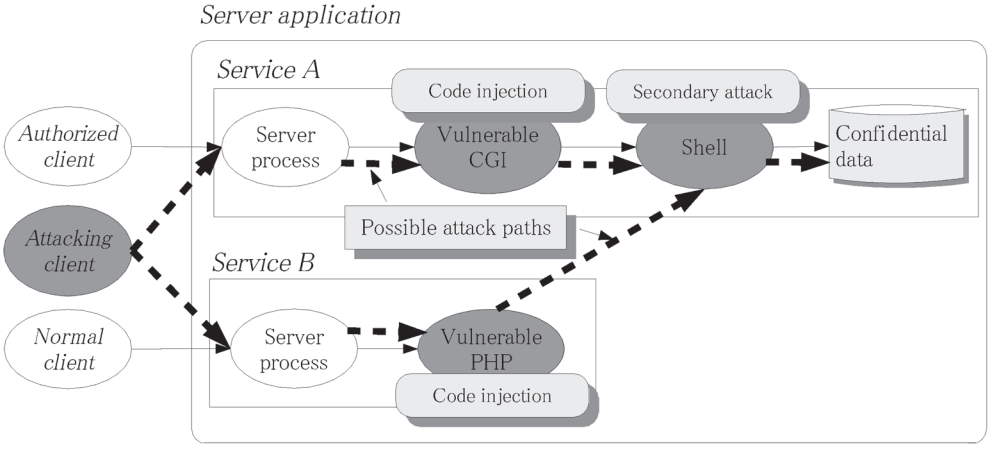

Fig. 1 Example of attacks using code injection of vulnerable processes.

correspond to process creation events such as `fork(2)`, `execve(2)`, etc. Event nodes correspond to other system call events including `accept(2)`. An example behavior-tree is shown in **Fig. 2**.

The behavior-tree is updated every time any process of the server application issues a new system call event, so that the tree always holds the whole event history of the application. Nodes related to already finished sessions are discarded for memory efficiency.

The behavior-tree satisfies the following conditions:

- Condition 1: A process node can have child nodes if and only if the corresponding process actually generates corresponding events (including a process creation event).
- Condition 2: The order of child nodes holds the order of event occurrences.

Condition 1 guarantees that the parent-child relationship of process nodes corresponds to that of the processes, so that the descending node sequence to the current-event node from the server-process node corresponds to the process-execution sequence until the shell process generates the current event (see Fig. 2). The service where the current event occurs can be identified as a sequence of process execution.

Conditions 1 and 2 guarantee that an `accept(2)` event node on the left side of the current event node corresponds to the access request that triggers the current event (see Fig. 2). Since an `accept(2)` event has an IP address of a client as one of its event attributes, the client can be easily identified by the `accept(2)` event node.

Thus the behavior-tree represents behavior of the server application, and meets the requirements of comprehensiveness, traceability and identifiability. We describe the behavior-tree construction algorithm in Section 3.1.

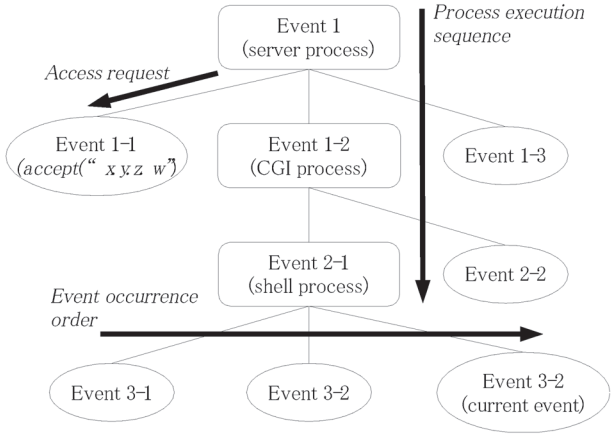

Fig. 2 Example behavior-tree (the rectangular nodes correspond to event creation events, and the oval nodes correspond to any other events such as file operations.)

2.2.2 Knowledge Description Model

To meet requirement (d), we develop a knowledge description model of the normal process behavior, called Context-Sensitive Policy (CSP). CSP can specify the relationship between the application-dependent specifications and each type of process behavior (i.e. a sequence of system calls).

CSP has three attributes: an identity context C, a service context S, and a process-behavior specification PB. CSP is formally denoted as follows:

$$CSP \triangleq (C, S, PB)$$

The identity context C specifies authorized clients, and is formally described as a set of IP addresses of the clients.

The service context S specifies a sequence of process-creation events. It is formally described as follows:

$$S \triangleq (\pi_0, \pi_1, ..., \pi_n),$$

where $\pi_i (1 \leq i \leq n)$ is a process invoked by its parent process π_{i-1}.

The process behavior specification is a set of system call events that a process is permitted to issue. It is formally described as follows:

$$PB \triangleq \{(syscall, P)\},$$

where *syscall* is a system call, and P is a set of resources.

Therefore CSP implies that if a currently observed process π is the last process π_n in S or one of its subprocesses, and if an access request sent by a member of C has triggered π, then the behavior of π is subject to PB.

From a system administrator's point of view, it is natural to consider a set of CSP's as a policy description for a server application, since a single CSP is too simple to adequately describe the behavior of a complicated server application. Hereafter we call each CSP a "CSP statement" (or simply "statement") and a set of statements a "CSP description" (or simply "description"). In a CSP description, each statement is evaluated in its order of occurrence. This evaluation

procedure is terminated the first time a new system call event meets a certain statement. Please note that if the event meets no statements, then the evaluation procedure is terminated and the event is forbidden by default.

We show an example of CSP description in **List 1**. It comprises two statements. The first statement is described in lines 1 to 3, where line 1 specifies an identity context that permits access requests from the site ".admin.com." Line 2 describes a service context that specifies a process execution sequence comprising a server process "/usr/sbin/httpd" and its subprocess "/var/www/cgi-bin/admin.cgi." Line 3 describes a process behavior specification that specifies a set of system call events to write files in the directory "/var/www/html" and its subdirectories, where the modifier "ALLOW" accepts the preceding event set as a process-behavior specification.

On the other hand, the second statement as described in line 5 forbids any other processes* to write files in the directory "/var/www/html" and its subdirectories, since the modifier "DENY" accepts a complementary set of the preceding event set as a process behavior specification.

Consequently the example description shown in List 1 permits only the administrators on the site ".admin.com" to maintain the Web contents allocated in the directory "/var/www/html" only if they use "admin.cgi," while users on unauthorized sites are forbidden to rewrite the Web contents. By such combination of CSP statements, CSP description can specify the relationship between application-dependent specifications and the process behavior of "admin.cgi."

2.2.3 Behavior-Based Process Confinement

Since the `accept(2)` event and the succeeding process-execution sequence related to the current event can be efficiently extracted from the behavior-tree, both the identity context *C* and the service context *S* are easily compared with the extracted event sequence. On the other hand, the process behavior specification *PB* is also easily compared with the current event. Thus each observed event is efficiently validated according to CSP policies, and any invalid event can be blocked as soon as it is detected.

We describe the details of the CSP matching algorithm in Section 3.2.

3. BEHAVIOR-TREE ALGORITHMS

3.1 Construction Algorithm

Basically a behavior-tree construction algorithm comprises the following steps:

1) Create an appropriately typed node (i.e. process node or event node) of a newly observed event. The node and the event are called the current node and the current event, respectively.
2) Find an appropriate parent process node of the current node.
3) Attach the current node to the process node as its rightmost child.

In step 1), the current node attributes are also generated. The attributes include the current event identifier (e.g. the system call name), its parameter values, and the process identifier (PID) of the process that generated the current event. Since general OS kernels assign a unique PID to each created process, the tree already has the corresponding process node with the PID. Therefore, step 2) can find the parent process node that has the same PID as that of the current node.† Steps 2) and 3) thus guarantee that behavior-trees generated by the algorithm always maintain the two conditions described in Section 2.2.

List 2 shows the complete construction algorithm,

The identity context "0.0.0.0/0" represents "any clients," and the service context "." represents "any processes."

†Strictly speaking, step 2) chooses the most recently created node among process nodes with the same PID in the tree.

```
1: .admin.com;\\
2: </usr/sbin/httpd></var/www/cgi-bin/admin.cgi>;
3: write,^/var/www/html/.*;ALLOW
4:
5: 0.0.0.0/0;.*;write,^/var/www/html/.*;DENY
```

List 1 Example of CSP policy description.

which constructs a corresponding behavior-tree from a given event sequence. Here we consider topological properties of the behavior-tree $T(\sigma)$ constructed from the event sequence σ beginning at the creation event of a server process π_0 (**Fig. 3**). In general, whenever a server application receives a new access request, its server process invokes an `accept(2)` system call to receive the request data. Then π_0 generates a new session, in which related tasks are performed by π_0 itself or by its subprocesses. Therefore, all nodes corresponding to events of each session must be descendants of the process node $n(\pi_0)$ corresponding to the server process π_0 in the behavior-tree $T(\sigma)$, so that $T(\sigma)$ has the following properties:

- Property 1: The node $n(\pi_0)$ corresponding to the server process π_0 is the root of $T(\sigma)$.
- Property 2: An accept(2) event node *accept*(r) generated when the application receives a request r is a child node of $n(\pi_0)$.
- Property 3: An event node *accept*(r) and all trees in $\tilde{T}(\sigma(r))$ appear successively as children of $n(\pi_0)$, where $\tilde{T}(\sigma(r))$ is a set of trees* corresponding to an event sequence $\sigma(r)$ that is generated in a particular session triggered by a request r.

These properties enable behavior-trees to have a modularized structure according to individual sessions, even if multiple sessions are processed by a server application in parallel (see Fig. 3 again). Therefore, a search space for the `accept(2)` event node related to the current node is limited to a local subtree, enabling us to develop the efficient CSP policy-matching algorithm described in the next section.

```
algorithm construct_behavior_tree
  input: σ : a sequence of events.
  output: T: T(σ) corresponding to σ.

  begin
    T := Φ ;
    while ( σ is not empty) do
    ev := dequeue(σ);
    if (ev is a process creation event)
    then m := create_process_node(ev);
    else m := create_event_node(ev);
    pid := get_process_id_of(ev);
    n := find_process_node(pid, T);
    if (n is not found)
    then T := T added m as an isolated node;
    else T := T added m as the rightmost child of n
    end;
    return T
  end.
```

List 2 Behavior-tree construction algorithm.

3.2 CSP Matching Algorithm

The CSP matching algorithm is composed of two parts: service context matching and identity context matching.

3.2.1 Service Context Matching

The matching algorithm for a service context comprises two steps: (a) extract a sequence of related process execution, and (b) match the sequence with a service context of a CSP policy. The first step extracts

*In this paper, we regard a single node as a tree.

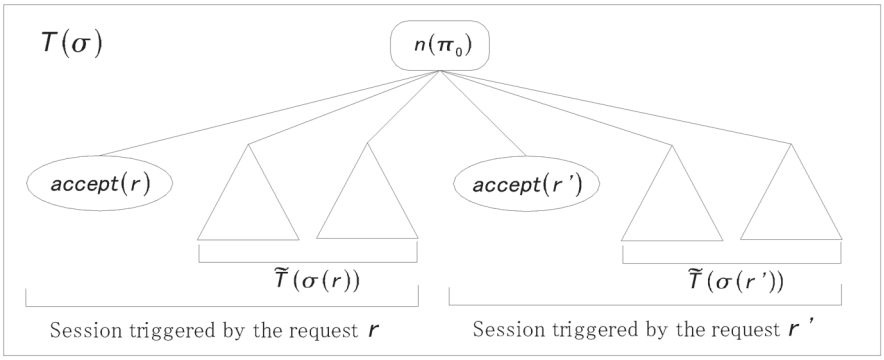

Fig. 3 Behavior-tree generated by a server application in successive sessions.

the sequence of process nodes by traversing a behavior-tree from the current node to the root node.* The reversed sequence corresponds to the process-execution sequence $\Pi = (\pi_0, ..., \pi_{i-1}, \pi_i)$ related to the current event. The second step compares Π to the service context of the policy using a string pattern matching method.

Assuming that the behavior-tree is approximately well balanced, the average computational cost of the first step is $O(\log(N))$, where N is the size of a subtree $\tilde{T}(\sigma(r))$ in since the length of Π equals the depth of the current node in the subtree. The computational cost of the second step is also $O(\log(N))$ because it is linear to the length of Π. Consequently the entire cost of evaluation is $O(\log(N))$.

3.2.2 Identity Context Matching

The matching algorithm for an identity context is composed of two steps: (a) search an `accept(2)` event related to the current node and (b) match an IP address of a corresponding client with the identity context. The first step searches the `accept(2)` event node by a procedure called clockwise traversal (see **List 3**). The procedure picks the `accept(2)` event node *accept*(*r*) that is nearest to the current node *n* on its left side.† The existence of such a node and the correctness of node selection are guaranteed by property 3. The second step compares an IP address of a client described in the `accept(2)` event node with the identity context using simple bit calculation. Obviously, the computational cost of the first step $O(N)$ is and that of the second step is of a constant order. Moreover, we can reduce the size of the behavior-tree by removing already validated event nodes from the tree (except `accept(2)` event nodes). This causes the complexity of the traversal procedure depend on only the tree's depth. The evaluation cost of the identity context can therefore be limited to $O(\log(N))$.

Finally, the entire computational cost of the CSP matching can be limited to $O(\log(N))$. Since N is the size of the subtree which holds the current node, the efficiency of policy matching is independent of the number of sessions. Furthermore, the logarithmic ordered matching cost is robust to long-term sessions, making the CSP matching algorithm suitable for large-scale server applications.

*The current node is not included in the sequence, even when it is a process node.

†If we can assume that a server process node is always the root of an arbitrary behavior-tree, there is a more straightforward method to find the `accept(2)` event node. However the assumption does not hold in general; thus the clockwise traversal is employed as a general method.

4. IMPLEMENTATION

4.1 Architecture

Based on the behavior-based confinement method, we have developed a new sandbox mechanism, called S-Tracer. S-Tracer comprises of the following components (**Fig. 4**):

- Interception module: This module intercepts system call events triggered by any process of the server application, and discards forbidden events detected by the detection module.
- Tracking module: This module constructs a behavior-tree from the intercepted events and mediates the interaction between the interception module and the detection module.
- Detection module: This module compares the current behavior-tree with given CSP policies, and detects irregular process behavior.

4.2 Implementation Details

S-Tracer is implemented as a loadable kernel module (LKM). Its interception module (1) hooks system

```
procedure clockwise_traversal
  input: T: the current behavior-tree,
         n: the current node.
  output: acc: the accept(2) event node.

  begin
    cn := n;
    pn := parent(cn);
    while (pn exists) do
      s := next_sibling_on_the_left_hand(cn);
      while (s exists) do
        if (s is an accept() event node)
        then do acc := s; return acc end;
        s := next_sibling_on_the_left_hand(s)
      end;
      cn := pn;
      pn := parent(pn)
    end;
    return OUT_OF_SESSION
  end.
```

List 3 Clockwise traversal procedure.

call handlers to intercept system call events generated by all processes, (2) passes each system call event to the detection module through the tracking module, and (3) prevents OS from performing of an illegal system call when the detection module detects its violation against a given CSP.

This implementation method contributes to satisfying the comprehensiveness requirement mentioned in Section 2.1. Some existing mechanisms are implemented in user space such as software wrappers [4,5]. Generally, these methods can be implemented more easily than LKM, and can monitor independent behavior of individual processes because they are embedded in each targeted process. Meanwhile, S-Tracer can monitor behavior of all processes because any system calls are processed by a single system call handler in which the interception module is embedded.

4.3 Application to a Server Security Solution

We have integrated S-Tracer into a security solution for Web applications, named "StarDefence." The main feature of StarDefence is to provide an integrated management environment for Web contents and server security.

This system uses S-Tracer as an attack detection component. In addition, as shown in **Fig. 5**, it consists of the following four components; (1) a management GUI, which provides Web-based GUI to server administrators; (2) an auditing module, which records Web content updates; (3) an access control module, which automatically performs emergency response against attacks; and (4) a logging module, which collects any alert generated by OS, libraries, S-Tracer, etc.

The management GUI enables server administrators to edit CSP descriptions for S-Tracer, to monitor logs and alerts, to maintain Web contents, etc. As for CSP editing, StarDefence can handle enhanced CSP statements, in which server administrators can describe ALLOW, DENY, WARN or RECOVER as process-behavior specifications; ALLOW and DENY have the same semantics as mentioned in Section 2.2. WARN specifies forbidden process behavior as does DENY. However, this does not imply that S-Tracer has the forbidden behavior interrupted, but only that it generates an alert that reports the occurrence of such behavior. Moreover, RECOVER makes the auditing module record an illegally modified content and then recovers the original one. Appropriate DENY specification ensures the prevention of damage; however, its misuse would damage availability of the applications. These enhancements, therefore, provide useful options that assist server administrators to be sure of Web applications availability.

The purpose of the access control module is, in collaboration with some firewall products such as Express5800/SG and ipchains, to interrupt continual attacks. Whenever S-Tracer detects that an attack has happened, this module immediately requests the previously designated firewall to block further access

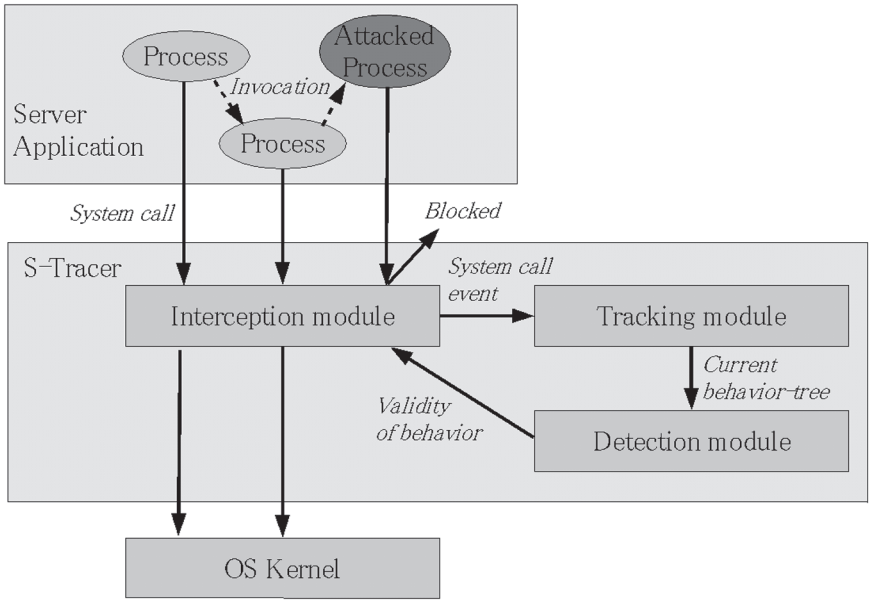

Fig. 4 Architecture of S-Tracer.

requests from the attacker's IP address detected by S-Tracer. This feature contributes especially to Internet worm prevention, since recent Internet worms tend to send malicious messages to the same server repeatedly.

Thanks to these features, StarDefence encourages both security and availability of Web applications. We would like to extend the target to general Internet server appliances in future.

5. EVALUATION

5.1 Effectiveness

In this section, we describe the effectiveness evaluation for CSP policies by performing pseudo attacks against an experimental Web application.

We first prepared the experimental Web applications comprising Apache 2.0.27 Web server daemon and two CGI applications on Linux 2.4.18. One of the CGI applications provides a personal information registration service for external network domains ("register-pl.cgi"), while the other provides a browsing service of the registered information for administrators on an authorized network domain ("view-pl.cgi"). We implemented the CGI applications in Perl, and embedded command injection vulnerability in them intentionally.

Next we prepared S-Tracer with CSP policies specifying the normal behavior of the daemon process, the CGI processes, and subprocesses of the CGI processes.

We then developed six types of pseudo attack code with the following functions implemented:

① Illegal modifications of the password file ("/etc/passwd").
② Illegal modifications of the personal information database ("/var/lib/present2003/info.csv").
③ Illegal references to the password file.
④ Illegal references to the personal information.
⑤ Illegal installation of a backdoor daemon listening to the port 38129.
⑥ Illegal installation of a Trojan process sending the application data contents to a remote server.

We encoded the six attack codes into the twelve URL strings for the two CGI applications, and then infected the Web application with the attack codes from the external network. As a result, S-Tracer successfully prevented all attacks without actual harm.

For example, all attacks against "view-pl.cgi" were blocked immediately after the CGI process had been invoked, since the access request was sent from an unauthorized domain (i.e. the identity context violation). On the other hand, the illegal modification attack against "register-pl.cgi" (No.2 in the above list) was blocked before the application data was modified. This is because a shell process invoked by "register-pl.cgi" directly tried to modify the application data, though the normal behavior of the shell process was supposed to send a thank-you mail to the registered person (i.e. the service context violation).

The result shows the effectiveness of S-Tracer against illegal code injection attacks. We also confirmed that S-Tracer could prevent some in-the-wild exploits such as "OpenFuck"[6] and "Wu-ftpd common vulnerability attack"[7].

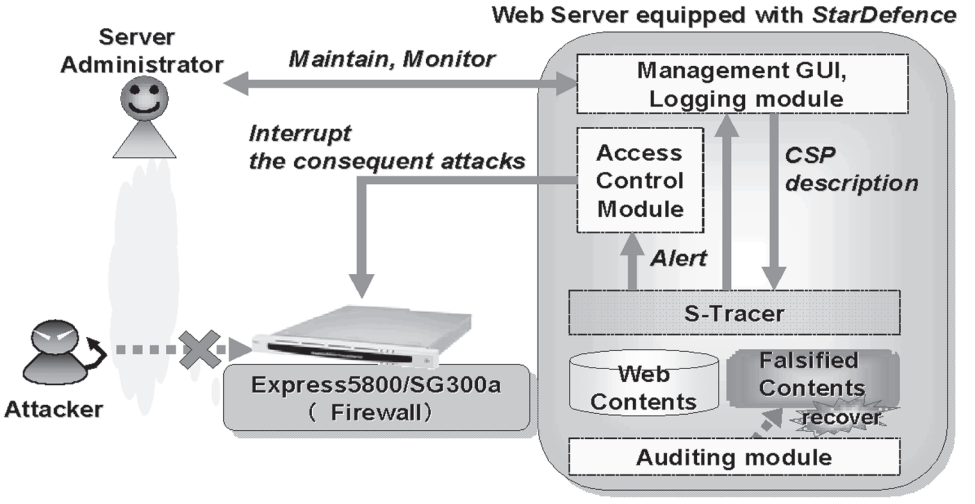

Fig. 5 Brief structure of StarDefence.

5.2 Performance

In this section, we explain the performance evaluation for S-Tracer by simulating typical network traffics. This simulation is based on an access log recorded by a Web server that actually runs in an Internet service provider.

The experimental server has a Pentium 4 2GHz processor and 1GB of memory, while the client has a Pentium 3 1GHz processor with 512MB of memory. They are connected by 100BASE-TX (Full-Duplex). The access log is for 24 hours and has 4,350,932 records. Its minimum, average and maximum access rates are 299 accesses per minute, 3021.5 and 6943 respectively. In order to simulate such network traffic, we have developed an automatic access tool in Java that sends HTTP requests at the same access rate as the given access log, and made it run on the client for 24 hours.

In this experimental system, we recorded transitions of CPU load, network throughput and turnaround time of the server. The results are shown in **Table I**. Obviously, S-Tracer had no influence on the network performance, and it increased the server CPU load by a paltry 0.5%. We also confirmed that the memory consumption of S-Tracer was limited within 120KB. From these results, we can conclude that the performance of S-Tracer is practically sufficient.

5.3 Administrative Workload

In this section we describe the user test for evaluating a workload for server administrators in order to define a CSP description for their Web application. This evaluation is performed in cooperation with a professional security administrator of a security service department. He has been engaging in security administration for three years, and has the necessary knowledge and usage experience for Linux OS, debugging tools, Web applications and security tools such as firewalls and network intrusion-detection systems (NIDS).

Table I Results of the performance evaluation.

	a) without S-Tracer	*b) with S-Tracer*	*Overhead (= b– a)*
Avg. CPU load (%)	2.93	3.43	0.50
Avg. throughput (KB/sec)	209.79	209.79	0.00
Avg. turnaround (msec)	0.0011	0.0010	–0.0001

First of all, we developed a small Web application comprising Apache Web server 2.0.27, some static Web contents and a Perl CGI module. The CGI module performs keyword searches for the static Web contents, but is intentionally embedded a direct OS command injection vulnerability in its keyword interpretation procedure.

Next we prepared an instruction guide explaining S-Tracer usage, the CSP syntax and format, and a standard procedure for CSP definition stipulated as follows:

① Investigate file I/O's of the Apache and the CGI module thoroughly.
② Describe a primary CSP description based on the results acquired in step ①.
③ Enforce the current CSP description to S-Tracer and perform a preliminary test to the Web application.
④ If you observe incorrect (i.e. false-positive) alerts by S-Tracer, then add some CSP statements as necessary to reduce the false-positive alerts into the current CSP description.
⑤ Repeat steps ③ and ④ until no false-positive alerts occur.

The testee then defined a CSP description for the experimental Web application by performing the above procedure.

Finally the testee was able to define the CSP description with no false-positive. The procedure lasted 1.5 hours: 1 hour for step ① and 0.5 hours for steps ② - ⑤ (steps ③ and ④ were repeated five times). The CSP description included 22 statements. By reviewing this description, we confirmed that the description successfully covers normal behavior of the experimental application. In the post-experiment interview with the testee, he said he estimated an average workload to define a CSP description for a general Web application at 3 hours, and concluded that this workload is permissive in most cases.

6. CONCLUSION

We have presented a behavior-based process confinement method using context-sensitive policies (CSP). CSP can describe the relationship between event sequences of a server application and application-dependent specifications.

In order to efficiently match CSP with event sequences of the server application, we also developed Behavior-Tree, a new application-behavior model.

The behavior-tree always maintains the event sequences including both an access-request event and a process-execution sequence which are related to each session of the server application. Based on the model, we developed an efficient CSP matching algorithm.

We have also developed S-Tracer, a sandbox mechanism based on this method. This engine can prevent irregular process behavior that is not authorized in the application-dependent specifications. Thus, the engine can efficiently protect potentially vulnerable server applications. We have then applied S-Tracer to StarDefence, a security solution for prevention of Web contents falsification.

Through the effectiveness evaluation test, we have confirmed that the engine can prevent falsification, theft of confidential data, and backdoor creation resulting from code injection attacks against vulnerable CGI processes. As a result of the performance evaluation and the user test, we have also confirmed that the kernel space implementation is promising and that the workload to define a CSP description is permissible in conventional administrative operations.

We can conclude that S-Tracer and StarDefence achieve high practicality and effectiveness in order to keep Web applications secure. In future, we would like to expand the scope of this technology for more general server products.

REFERENCES

[1] C. Cowan, C. Pu, et al., "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," Proc. 11th USENIX Security Symposium, 2002.

[2] V. Kriansky, D. Bruening, et al., "Secure Execution via Program Shepherding," Proc. 11th USENIX Security Symposium, 2002.

[3] S. Chari, and P. C. Cheng, "BlueBoX: A Policy-driven, Host-Based Intrusion Detection System," ACM Transactions on Information and System Security (TISSEC), **6**, 2, 2003.

[4] D. Peterson, M. Bishop, et al., "A Flexible Containment Mechanism for Executing Untrusted Code," Proc. 11th USENIX Security Symposium, 2002.

[5] A. Acharya, and M. Raje, et al., "MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications," Proc. 9th USENIX Security Symposium, 2000.

[6] CAN-2002-0656, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0656.

[7] Multiple Vendor Common Vulnerability, http://www.securityfocus.com/bid/2240/info/.

Received December 1, 2004

* * * * * * * * * * * * * * * *

Masayuki NAKAE received his M.E. degree from Osaka University in 1995. He joined NEC in 1995 and is now Assistant Manager of the Internet Systems Research Laboratories. He is engaged in the research and development of system security technologies.

Ryuichi OGAWA received his M.S. degree in Geophysics from Tokyo University in 1983. He joined NEC in 1983 and is now Principal Researcher of the Internet Systems Research Laboratories. He is engaged in the research and development of system security technologies.

Yasushi SATO received his B.S degree in Tokyo Institute of Technology in 1994. He joined NEC in 1994 and is now Assistant Manager of the System Platform Software Development Division. He is engaged in the research and development of system security solutions.

Sonomi SHIOZAWA received her master's degree in Science and Engineering at University of Tsukuba in 1991. She joined NEC in 1991 and is now Assistant Manager of the System Platform Software Development Division. She is engaged in the research and development of system security solutions.

* * * * * * * * * * * * * * * *