2004

Context Sensitive Anomaly Monitoring of Process Control Flow To Detect Mimicry Attacks and Impossible Paths

Haizhi Xu
Syracuse University, Systems Assurance Institute

Wenliang Du
Syracuse University, Systems Assurance Institute, wedu@syr.edu

Steve J. Chapin
Syracuse University

Context Sensitive Anomaly Monitoring of Process Control Flow to Detect Mimicry Attacks and Impossible Paths*

Haizhi Xu, Wenliang Du, and Steve J. Chapin

Systems Assurance Institute, Syracuse University, Syracuse NY 13244, USA
{hxu02, wedu, chapin}@ecs.syr.edu

Abstract. Many intrusions amplify rights or circumvent defenses by issuing system calls in ways that the original process did not. Defense against these attacks emphasizes preventing attacking code from being introduced to the system and detecting or preventing execution of the injected code. Another approach, where this paper fits in, is to assume that both injection and execution have occurred, and to detect and prevent the executing code from subverting the target system. We propose a method using waypoints: marks along the normal execution path that a process must follow to successfully access operating system services. Waypoints actively log trustworthy context information as the program executes, allowing our anomaly monitor to both monitor control flow and restrict system call permissions to conform to the legitimate needs of application functions. We describe our design and implementation of waypoints and present results showing that waypoint-based anomaly monitors can detect a subset of mimicry attacks and impossible paths.

Keywords: anomaly detection, context sensitive, waypoint, control flow monitoring, mimicry attacks, impossible paths

1 Introduction

Common remote attacks on computer systems have exploited implementation errors to inject code into running processes. Buffer overflow attacks are the best-known example of this type of attacks. For years, people have been working on preventing, detecting, and tolerating these attacks [1–13]. Despite these efforts, current systems are not secure. Attackers frequently find new vulnerabilities and quickly develop adaptive methods that circumvent security mechanisms.

Host-based defense can take place at one of three stages: preventing code injection, preventing execution of the injected code, and detecting the attack after the injected code has begun execution. One class of detection mechanisms, execution-monitoring anomaly detection, compares a stream of observable events in the execution of a running process to a profile of "known-good" behavior, and raises alerts on deviations from the profile. While it is possible to treat each instruction executed by the process as an event

* This work was supported in part by a Syracuse University Graduate Fellowship Award.

for comparison to the profile, typical anomaly detectors use system calls [6, 14–17] or function calls [5, 18] as the granularity for events.

We focus our efforts on detecting attempts to subvert the system through the kernel API (the system call interface), assuming the attacking code has started to run. We monitor requests for system services (i.e., system calls) of running processes, and detect anomalous requests that could not occur as a result of executing the code in the original binary program image.

Two major problems that system-call based anomaly detection faces are mimicry attacks [12, 19] and impossible paths [12]. A mimicry attack interleaves the real attacking code with innocuous code, thereby impersonating a legitimate sequence of actions. For example, if the legitimate code has the system call sequence

```
getuid() ... open() ... execve()
```

and the attack has the sequence

```
getuid() ... execve()
```

the attacker can add a "no-op" system call to match the legitimate attack sequence:

```
getuid() ... open("/dev/null"...) ... execve().
```

We further divide mimicry attacks into global mimicry attacks and local mimicry attacks. Considering the minimum set of system calls necessary for the functionality of an application function, the system call sequence in a global mimicry attack combines the legal system calls of multiple functions, while a local mimicry attack uses the legal system calls of only the running function.

An impossible path is a control path (with a sequence of system calls) that will never be executed by the legal program, but is a legal path on the control flow graph of the program. Impossible paths can be generated due to the nature of the non-deterministic finite state automata (NDFSA). For example, when both location `A` and `B` can call function `f()`, function `f()` can return to either location A or B. The call graph for the program allows a return-into-others impossible path wherein location `A` calls function `f()`, but the return goes to location `B`, which behavior appears legal in the control flow graph. This example attack is similar to a return-into-lib(c) attack in that both of them modify the legal control path at the function return points.

This paper introduces our use of waypoints[1] for assisting anomaly monitoring. Waypoints are kernel-supported trustworthy markers on the execution path that the process must follow when making system calls. In this paper, we use function-level scoping as the context of a waypoint. If function `C` calls function `D`, then the active context is that of `D`; upon function return, the active context is again that of `C`. Waypoints provide control flow context for security checking, which supports call flow checking approaches such as that in Feng, et al. [5] and allows us to check whether the process being monitored has permission to make the requested system call in the context of the current waypoint.

The work presented in this paper makes the following contributions:

[1] This terminology is borrowed from route planning using a GPS system, while the term is as old as navigation systems, meaning a specific location saved in the receiver's memory that is used along a planned route.

1. Kernel-supported waypoints provide fine-grained trustworthy context information for on-line monitoring. Using this information, we can restrict a process to access only those system calls that appeared in the original program fragment associated with the waypoint context.
   Waypoints can change the granularity of intrusion detection systems that monitor system call sequences. The more waypoints we set between two system calls, the more precise control of that program path we can provide to the detector.
2. Using the context information, our anomaly monitor can detect global mimicry attacks that use permissions (i.e. allowed system calls) across multiple functions. Any system service request falling out of the permission set of the current context is abnormal.
3. Our anomaly monitor can detect return-into-others impossible paths attacks. We use waypoints to monitor the function call flow and to guarantee that callees return to the right locations.

In the next section, we describe our model of attacks in detail. Section 3 describes our design and implementation of waypoints and the waypoint-based system call monitor. In section 4 we present performance measurements of our approach. Section 5 summarizes related work. Section 6 discusses the limitations and our future work and gives our conclusions.

2 Attack Models

Once the exploit code has a chance to run, it can access the system interface in the following three ways, which we present in order of increasing code granularity:

1. Jumping to a system call instruction, or a series of such instructions, within the injected code itself. Many remote attacks use shellcode—a piece of binary code that executes a command shell for attackers [20]. Most shellcode issues system requests directly through pre-compiled binary code. In this case, the attacker relies on knowing the system call numbers and parameters at the time he compiles the attack code, which, in the presence of a near monoculture in system architecture and standardized operating systems, is a reasonable assumption. The control path in this case is fully under the control of the attacker, as he controls the location of the sensitive system call at the time of the code injection.
2. Transferring control to legitimate code elsewhere in the process; the target code can be at any link on the path to the system call instruction. The attacking code achieves its goal by executing from the target instruction forward to the desired system call. The attack can be achieved either by creating fake parameters and then jumping to a legitimate system call instruction, (e.g., making it appear that the argument to an existing `execve()` call was `"/bin/sh"`), or by jumping to a point on the path leading to an actual call (e.g. `execve("/bin/sh")`) in the original program. Locations used in the latter attack include a system call wrapper function in a system library such as `libc`, an entry in the procedure linkage table (PLT) using the address in the corresponding global offset table (GOT), or an instruction in an application function that leads to calling the above entries. This is the general form

of the `return-into-lib(c)` attack [21], in which the corrupted return address on the stack forces a control transfer to a system call wrapper in the `libc` library. For the remainder of this paper we will refer to this type of attack as an low-level control transfer (LCT) attack. In contrast to defending against the shellcode attack, it is of paramount importance to protect the control path when defending against an LCT attack.

3. Calling an existing application function that performs the system call(s) that the attacking code requires. While this is a form of control transfer attack, we distinguish it from the LCT because the granularity of the attack is at the application function level, not at the level of the individual instruction or system call. In this case, the control path is the sequence of application-level function invocations leading to the function that contains the attacking call.

Mimicry attacks can be achieved by directly jumping to injected code that mimics a legal sequence of system calls or calling a sequence of lib(c) functions, which fall in the above category 1 and 2 attacks. Attackers can also use category 3 attacks (i.e. calling existing application functions), but this is easier to detect than the category 1 and 2 attacks by using call flow monitoring techniques. Attackers can also explore impossible paths to elude detection by using the above three categories attacking techniques.

While function call flow monitoring can reduce attacks in category 3, and non-executable data sections can block attacks in category 1, attacks using category 2 techniques are more difficult to detect because they use legitimate code to achieve malicious purposes. An important characteristic that attackers use is that the default protection model permits programs to invoke any system call from any function, but in actuality each system call is only invoked from a few locations in the legal code. While some previous work has exploited the idea of binding system calls or other security sensitive events with context [5, 18, 22–24], this paper explores this approach further. We introduce the concept of waypoints to provide trustworthy control flow information, and show how to apply the information in anomaly detection.

3 Waypoint-based System Call Access Control

We observe that an application function—a function in an application program, not a library function—in general uses only a small subset of system service routines[2], but has the power to invoke any system call under the default Unix protection model. This practice violates the principle of least privilege, which restricts a function to only invoke systems calls that are necessary for its proper execution. For example, `execve()` is not used by many legitimate functions, especially in setuid root regions, but it is common for exploit code to invoke that system call within the scope (or equivalently, the stack frame) of any vulnerable function. Waypoints provide a mechanism for restricting program access to system calls and enforces least privilege.

[2] For example, in table 1 and 2 of section 3.3, only 3 out of 416 application functions being monitored require `execve()` legally.

3.1 Waypoint Design

A waypoint, located in the code section, is a trustworthy checkpoint on control flow. Waypoints can actively report control flow information in real-time to assist intrusion detection, rather than gathering the information only at system call time. People can assign security attributes to each waypoint or to a sequence of waypoints.

To achieve our goals, waypoints must embody the following properties:

1. Authentication
   Because we assume that an attack has successfully started executing, and the attack has the right to access the whole process image, it is possible that the attacking code can overwrite code pointers. Although the code section is usually read-only, dynamically-generated code will be located in memory with both read and write permissions. This means that attackers have the ability to generate waypoints within their own code, and we must therefore authenticate waypoints.
   We authenticate the waypoints by their locations. Waypoints are deployed before the process runs, such that the waypoint locations are registered at program loading time. In this way, we can catch false waypoints generated at run time.
2. Integrity
   Because attackers can access the whole process image, information generated at and describing waypoints (e.g., their privileges) should be kept away from malicious access. We store all waypoint-related data and code in the kernel.
3. Compatibility
   Our waypoints work directly on binary code, so the original code may be generated from different high-level languages or with different compilers.

A natural granularity for control flow monitoring is at the function level. To trace function call flow, we set up waypoints at function entrance and exit. We generate waypoints and their associated permissions on a per-function basis through static analysis.

At run time, we can construct a push-down automata of the waypoints that parallels the execution stack of the process. An entrance waypoint disables the permissions of the previous waypoint, pushes the entrance waypoint on top of the waypoint stack, and enables its permissions. A corresponding exit waypoint discards the top value on the waypoint stack and restores the permissions of the previous waypoint.

It is possible that we assign different permissions to different parts of a function. In this case, we need a middle waypoint. A middle waypoint does not change the waypoint stack. It only changes permissions for the waypoint.

We deploy waypoints only in the application code. Although we do not set waypoints in libraries, we are concerned about library function exploitation. We treat security relevant events (system requests) triggered by library functions as running in the context of the application function that initiated the library call(s).

The waypoint stack records a function call trace. Using this context information, our access monitor can detect attacks in two ways: (1) flow monitoring—Globally, waypoints comprise the function call trace for the process. We can construct legal waypoint paths for some security critical system requests (e.g. `execve()`), such that when such a system call is made, the program must have passed a legal path. Similar ideas on control flow monitoring have been proposed in [5, 25], therefore, we do not discuss this

approach further in this paper. (2) permission monitoring—Locally, we use static analysis to determine the set of system calls (permissions) required for each function, and ensure system calls invoked in the context of a function appears in its permission set.

3.2 Waypoint Implementation

If a piece of code needs to perform a system request legally, then we say that the piece of code has a permission to issue the system request. To simplify the implementation, we use a set to describe permissions for a waypoint and store the permission sets in a bitmap table.

We generate waypoints and their corresponding permissions through static analysis. We introduce global control flow information by defining the number of times that a function can be invoked. Usually, an application function does not issue system requests directly. It calls system call wrappers in the C library instead. The application may call the wrapper functions indirectly by calling other library functions first. We build a (transitive) map between system call wrappers and system call numbers. Currently, we analyze the hierarchical functions manually. Our next step is to automate this whole procedure.

We deploy the access monitor, together with the waypoint stack and the permission bitmap table, in the operating system kernel, as shown in figure 1. There are two fields in an entry of the waypoint stack, one is the location of the waypoint, the other is extra information for access monitoring. Since we monitor application function call flow, we use this field to store the return address from the function. In one application function, there is one entry waypoint and one exit waypoint, the pair of which is stored in the bitmap table. Field "entries" in the bitmap table indicates how many times a waypoint can be passed. In our current implementation, we only distinguish between one entry and multiple entries to avoid malicious jump to prologue code and function `main()`, which usually contain some dangerous system calls and should be entered only once.

At a waypoint location, there should be some mechanism to trigger the waypoint code in the kernel. We can invoke the waypoint code at several locations: an exception handler, an unused system call number service routine, or a new soft interrupt handler. We insert an illegal opcode at the waypoint location and run our waypoint management code as an exception handler.

An attacker can overwrite the return address or other code pointer to redirect control to a piece of shellcode or a library function. We protect the return address by saving it on the waypoint stack when we pass the entrance waypoint. When a waypoint return is executed at the exit waypoint, the return address on the regular stack is compared with the saved value on the waypoint stack for return address corruption. The exit waypoint identifier must also match with the entrance waypoint identifier, since they come in pairs. If the attacking code uses an unpaired exit waypoint or a faked waypoint, the comparison will fail. If the attack forces return into a different address, although the control flow can be changed, the active permission set—the permission set belonging to the most recently activated waypoint—is not changed, because the expected exit waypoint has not passed. The attacking code will still be limited to the unchanged permissions.

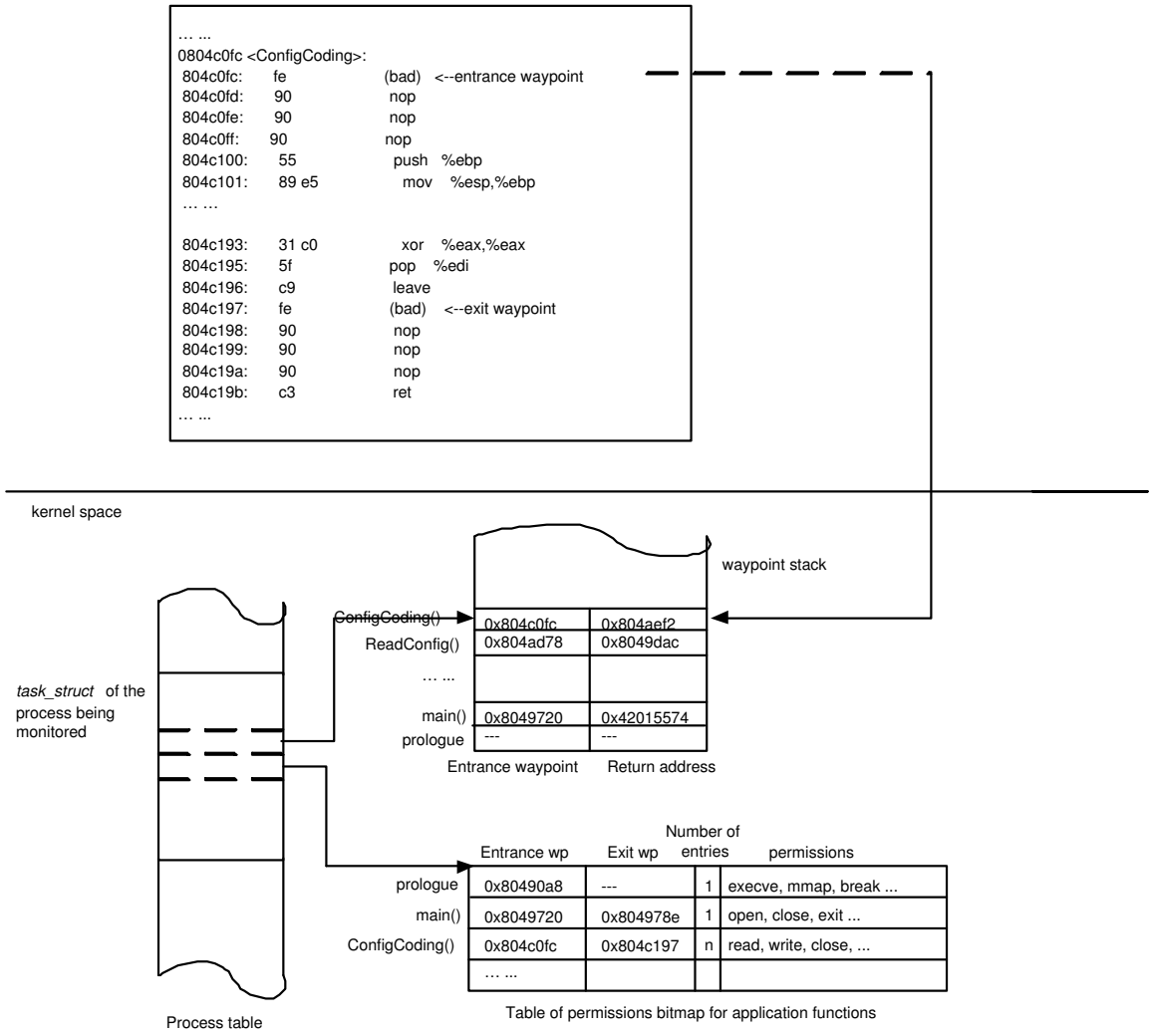

Fig. 1. Data structures needed for the waypoint-based access monitor: a waypoint stack and a table of permission bitmaps. The third column of the bitmap table indicates how many times a waypoint may be activated. The prologue code and function main() are allowed to run only one time during the process life. Function ConfigCoding() can be called unlimited times.

3.3 Monitoring Granularity

In our implementation, each waypoint causes a kernel trap, and each guarded function has at least two waypoints (an entrance/exit pair, plus optional middle waypoints). Thus, the performance of the system is dependent on the granularity of waypoint insertion. Our first implementation monitored every function, irrespective of whatever system calls the function contained. As reported in section 4, the overhead can be substantial.

Not all system calls are equally useful for subverting a system. We define dangerous system calls as those rated at threat level 1 in [26]. There are 22 dangerous system calls in Linux: `chmod`, `fchmod`, `chown`, `fchown`, `lchown`, `execve`, `mount`, `rename`, `open`, `link`, `symlink`, `unlink`, `setuid`, `setresuid`, `setfsuid`, `setreuid`, `setgroups`, `setgid`, `setfsgid`, `setresgid`, `setregid`, and `create_module`. Such system calls can be used to take full control of the system.

Table 1 and 2 show the number of functions containing dangerous system calls, and the permissions distribution. The tables show us that only a small portion of the application functions invoke dangerous system calls. Most functions call at most one dangerous system call, and no function calls more than three. Only three functions (two in `tar` and one in `kon2`) in the whole table require `exec`. Most functions invoking three dangerous system calls contain only file system-related calls such as `open`, `symlink`, and `unlink`.

program	# of application functions totally	# of functions containing dangerous system calls	containing 3 dangerous system calls	containing 2 dangerous system calls	containing 1 dangerous system calls
enscript	48	8	0	0	8
tar	165	26	3	3	20
gzip	92	6	1	3	2
kon2	111	10	1	5	4
totally	416	12%	1.2%	2.6%	8%

Table 1. number of functions invoking dangerous system calls, and the calls distribution. Only 12% (50) of functions in our analysis use dangerous system calls, while 1.2% (5) of them contains 3 dangeous calls.

The distribution of dangerous system calls shows that partitioning of system call permissions should be effective. If an exploit happens within the context of one function, the attacker can use only those system calls authorized for that function, which significantly restricts the power of attack in gaining control of the system. Existing code-injection attacks exploit flaws in input routines, which do not, in general, have permission to call dangerous system calls. `Open`, however, is used widely in application functions (9% of application functions use it), requiring further restrictions to its parameters.

program	# of functions	create_module	execve	setu(/g)id	open/rename	chown(/mod)	link, sym(/un)link
enscript	48	0	0	0	8	0	0
tar	165	0	2	1	20	3	4
gzip	92	0	0	0	3	2	5
kon2	111	0	1	1	6	4	2
totally	416	0	0.7%	0.5%	9%	2.2%	2.6%

Table 2. number of functions invoking dangerous system calls. For example, 20 functions in `tar` invokes `open` or `rename`. Only 0.7% (3) of all functions in our analysis call `execve`.

The numbers show that as an alternative of monitoring every function, we can monitor only functions containing dangerous system calls to detect subversions. In this case, we have a default permission set that allows all other system calls, and only deploy waypoints when switching between the general, default permissions and the strict, specific permissions associated with the function that uses dangerous system calls. This is a conscious trade-off of capability for performance; we no longer have a full waypoint stack in the kernel that reflects all function calls during program execution, but the overhead decreases significantly, as shown in figure 3 of section 4.

3.4 Waypoint-based Anomaly Monitoring

After generating waypoints and their permission sets for a program, we monitor the program at run time. The procedure of the waypoint-based security monitor can be described in the following steps:

1. Marking a process with waypoints
   At the process initialization stage, we mark a process by setting a flag in its corresponding `task_struct` in the process table, indicating whether the process is being monitored or not. For a process being monitored, we set up a waypoint stack and create a table of permission bitmaps for the waypoints. The permission sets are generated statically.
2. Managing the waypoints at run time
   Waypoints are authenticated by their linear addresses. We implement the management procedure in an exception handler. When an exception is triggered, we first check whether it is a legitimate waypoint or not. A legitimate waypoint satisfies three conditions: (1) the process is being monitored; (2) the location of the exception (waypoint location) can be found in the legal waypoint list; and (3) the number of times that the waypoint is activated is less than or equal to the maximum allowed times. If the conditions are not satisfied, we pass control to the regular exception handler.

After the verification, we manage the waypoint stack according to the type of the waypoint. If it is an entrance waypoint, we push it onto the waypoint stack and activate its permission set; if it is a middle waypoint, we only update the permissions; and if it is an exit waypoint, we pop the corresponding entrance waypoint from the stack and restore the previous permission set. After that, we emulate the original instruction if necessary, adjust the program counter to the location of the next instruction and return from the exception handling. To simplify implementation, we insert 4 `nop`s at the waypoint locations and change the first `nop` to a waypoint instruction (i.e. a bad instruction in our implementation). In this way, we can avoid emulating the original instructions, because `nop`s perform no operations.

3. Monitoring system requests
   We implemented the access monitor as an in-kernel system call interceptor in front of the system call dispatcher. In terms of access control logic, the subject is the application function; the object is the system call number; and the operation is the system call request. After trapping into the kernel for a system call, the access control monitor first verifies whether the current process is being monitored or not. If yes, the monitor fetches the active waypoint from the top of the waypoint stack and its corresponding permission set from the permissions bitmap table. If the request belongs to the permission set, the monitor invokes the regular system service routine; otherwise, the monitor refuses the system call request and writes the violation information in the kernel log.

3.5 Implementation Issues

We have considered the following issues in our implementation:

1. monitoring offspring processes
   We monitor the offspring processes the same way as we monitor the parent process. A child process inherits the monitor flag, the permission bitmap table, the waypoint stack, and the stack pointer from the parent process. If the child is allowed to run another program (e.g. by calling `execve()`), then the waypoint data structures of the new program will replace the current ones.
2. multiple-thread support
   Linux uses light-weight processes to support threads efficiently. Monitoring a light-weight process is similar to monitoring an ordinary process, but requires a separate waypoint stack for every thread. Our current implementation does not support thread-based access monitoring.
3. number of passes
   By restricting the number of times a waypoint can be passed during a process life time, we can monitor some global control flow characteristics efficiently. In particular, we allow the program prologue to start only one time, because it typically invokes dangerous system calls and is logically intended to run only once. We also allow `main()` to start only once per process execution.
4. non-structured control flow
   Control flow does not always follow paths of function invocation. In the C/C++ languages, the `goto` statement performs an unconditional transfer of control to

a named label, which must be in the current function. Because `goto` does not cross a function boundary, it does not affect function entrance and exit waypoints. However, it might jump across a middle waypoint, so we do not put any middle waypoints between a goto instruction and the corresponding target location.
`Setjmp` sets a jump point for a non-local goto, using a `jmp_buf` to store the current execution stack environment, while `longjmp` changes the control flow with the value in such a data structure. At the `setjmp` call, we use a waypoint to take a snapshot of the in-kernel waypoint stack and the `jmp_buf`, while at the `longjmp` location, a waypoint ensures that the target structure matches a `jmp_buf` in the kernel, and replaces the current waypoint stack with the corresponding snapshot.

5. permissions switch with a low-overhead policy
   Under our low-overhead policy, we only monitor functions that invoke dangerous system calls. These functions may call one another, or call a function with only default permissions, or vice versa. Permissions are switched on the function call boundary. In the forward direction, where the caller has specific, elevated permissions, we use a middle waypoint to switch to the default permission set before calling, and switch the permissions back after returning. If the callee has specific, elevated permissions, regular entrance and exit waypoints will activate them.
6. the raw system interface
   To control the target system, an attacker may use the raw system interface (e.g. `/dev/mem` and `/dev/hda`). This is an anomaly to most applications. Our waypoint-based defense will restrict the opportunities for the attacker to call `open`, but further defenses, e.g. parameter checking, are necessary for complete defense. Our current implementation does not employ parameter checking. See [5, 25–27] for further information on parameter monitoring.

3.6 Evasion Attacks and Defenses

Because the waypoint structures and code are located in the kernel, attackers cannot manipulate them directly. However, an adaptive attack may create an illegal instruction in the data sections as a fake waypoint or jump to the middle of a legitimate instruction (in an X86 system) to trigger the waypoint activation mechanism. As we explained in section 3.2, our waypoint management code can recognize the fake waypoints because all the legitimate waypoints are loaded into the kernel at load time. If an attack intentionally jumps over a waypoint, although it can change the control flow, the waypoint stack is not updated neither is the permission set.

Our waypoint mechanism was originally designed to counter attacks of category 1 (shellcode based attacks) and category 2 (LCT attacks) described in section 2, because these attacks bypass waypoints and therefore fail to acquire the associated permissions. Evasion attacks may use the category 3 attack (function granularity attacks), if these functions invoke the exact system calls and in the correct order, required by the attacker. For such programs, the low-overhead policy may not supply sufficient trace information to support function call flow monitoring, so full monitoring on function call path should be done.

If an attack launches a local mimicry attack, using one or a sequence of legitimate system calls of the current context, our mechanism cannot detect it. This is the general

case of abuse of the raw system interface mentioned above, and in similar fashion, we must employ complementary techniques. In our implementation, we adopt system interface randomization [2, 28] to counteract shellcode-based local mimicry attacks.

Existing implementations of system call number randomization [2] uses a permutation of the system call numbers. A simple permutation of the relatively small space (less than 256 system calls) allows attackers to guess the renumbering for a particular system call in 128 tries on average, or 255 guesses in the worst case.

To survive this brute force attack, we use a substitution cipher to map from 8-bit system call numbers to 32-bit numbers, thereby making a brute-force attack on the system impractical. In Linux, a system call number n is an unsigned 8-bit integer between 0 and 255, and is carried to the kernel in register `%eax`, a 32-bit register, of which 24 bits are unused. In our implementation, we make use of the whole register to carry the 32-bit system call number. We generate a one-to-one mapping between the 8-bit system call numbers and their corresponding 32-bit secrets. The access monitor restores the original number correspondingly upon a system call.

3.7 An Example

To demonstrate the effectiveness of our waypoint mechanism, we attacked a real application program in Linux, using both shellcode and return-into-lib(c) attacks. We chose `kon2` version 0.3.9b as the target. `kon2` is a Kanji emulator for the console. It is a setuid root application program. In version 0.3.9b, there is a buffer overflow vulnerability in function `ConfigCoding()` when using the `-Coding` command line parameter. This vulnerability, if appropriately exploited, can lead to local users being able to gain root privileges [29]. Part of the source code of the vulnerable function `ConfigCoding()` is shown in figure 2(a), with the vulnerable statement highlighted. Figure 2(b) shows its original binary code, and figure 2(c) shows the binary code with waypoints added.

To help the shellcode attack reach our waypoint mechanism, we disabled the system call renumbering and return address comparison features of our system during our experiment. In the following attack and defense experiment, we show how the waypoint mechanism can detect malicious system calls in both shellcode based and return-into-lib(c) based attacks.

- Attack 1: calling a system call instruction located in the shellcode
  In the attack, the return address of function `ConfigCoding()` is overflowed. In this experiment, the faked return address redirects to a piece of shellcode. Without our protection, the attacking code generated a shell. With our mechanisms deployed, the malicious system request `execve(``/bin/sh'')` was caught and the shell was not generated. At the location of the `ret` instruction, an exit waypoint is triggered, and the permissions for `ConfigCoding()`'s parent function (`ReadConfig()`) are activated. Because `execve()` is not among the permissions of `ReadConfig()`, the system request is denied. It is interesting to see that if the return address is overwritten, the malicious request is issued in the context of the parent function, because the malicious request is issued after the execution of

```
static int      ConfigCoding(const char *confstr)
{
   char reg[3][MAX_COLS]; <--Fixed size buffer  MAX_COLS=256
   int n, i;

   *reg[0] = *reg[1] = *reg[2] = '\0';
   sscanf(confstr, "%s %s %s", reg[0], reg[1], reg[2]);
   ^^^^^^^^^^^^^^^^^^^^^^^^^^buffer overflow vulnerability here


   ... ...
   return SUCCESS;
}
```

(a) A buffer−overflow vulnerable function in kon2

```
0804c0fc <ConfigCoding>:
 804c0fc:      90                 nop
 804c0fd:      90                 nop
 804c0fe:      90                 nop
 804c0ff:      90                 nop
 804c100:      55                 push   %ebp
 804c101:      89 e5              mov    %esp,%ebp
 ... ...

 804c193:      31 c0              xor    %eax,%eax
 804c195:      5f                 pop    %edi
 804c196:      c9                 leave
 804c197:      90                 nop
 804c198:      90                 nop
 804c199:      90                 nop
 804c19a:      90                 nop
 804c19b:      c3                 ret
```

(b) the original binary code

```
0804c0fc <ConfigCoding>:
 804c0fc:      fe                 (bad)   <--entrance waypoint
 804c0fd:      90                 nop
 804c0fe:      90                 nop
 804c0ff:      90                 nop
 804c100:      55                 push   %ebp
 804c101:      89 e5              mov    %esp,%ebp
 ... ...

 804c193:      31 c0              xor    %eax,%eax
 804c195:      5f                 pop    %edi
 804c196:      c9                 leave
 804c197:      fe                 (bad)   <--exit waypoint
 804c198:      90                 nop
 804c199:      90                 nop
 804c19a:      90                 nop
 804c19b:      c3                 ret
```

(c) the binary code with waypoints added

Fig. 2. a buffer overflow vulnerable function in kon2 and its waypoints

instruction `ret` and the exit waypoint. If our mechanisms are fully deployed, the exit waypoint will guarantee that the return address is not faked.

- Attack 2: A low-level control transfer attack
  Recall that a low-level control transfer attack can redirect control to legitimate code for malicious purposes. In our experiment, we use the location of `int execve (const char *filename, char *const argv[], char *const envp[])`, a sensitive libc function, in the attacking code. Because neither `ConfigCoding()` nor its caller `ReadConfig()` have the permission to call system call `execve()`, the request is rejected by our monitor.
  Note, it is difficult to detect the return-into-lib(c) attacks. Program shepherding [25] ensures that library functions are called at only library entrance locations, and the library callee functions must exist in the external symbol table of the ELF format program. In kon2, because `execl()` and `execlp()` are used at other locations, there are corresponding entries in the external symbol table; so at any library entrance point, this request can pass the shepherding check. In addition, program shepherding monitors control flow only, so it is possible for an attack to compromise control flow related data (e.g. GOT), making the return-into-lib(c) attack realistic. In an IDS without control flow information, because `execve()` is used in the program, a mimicry attack may pass the check.

The only dangerous system call in the context of `ConfigCoding()` is `open()`. Within this context, the attacker does not have much freedom in gaining control of the system. Launching an `execve()` requires a global mimicry attack that crosses function boundaries, which is subject to both the call flow and permissions monitoring.

4 Overhead Measurement and Analysis

We measured the overhead of the waypoint-based access monitor on a system of Red-Hat Linux 9.0 (kernel version 2.4.20-8) on a 800MHz AMD Duron PC with 256MB memory.

The overhead of the waypoint-based access monitor has two main causes: waypoint registration in the exception handler and running the access monitor at each system call. The system call mapping is done before running, so it does not introduce any run-time overhead. The remapping at each system call is a binary search on a 256 entry table in our implementation. Because the remapping takes only tens of instructions, this overhead is negligible. The access monitor at the system call invocation compares the coming request number with the permission bitmap. These comparison operations cost little time. Therefore, the majority of the overhead is from the additional trap for the waypoint registration code in the exception handler, where caches and pipelines will be flushed.

Our measurement on a micro-benchmark program that calls a monitored function in a tight loop shows that the overhead for one waypoint invocation is 0.395 microseconds on average. This captures the cost of exception handling, but does not reveal overhead due to cache and pipeline flushing.

To better understand these effects on real applications, we tested a few well known GNU applications. We did not use real time, the time between program start and end,

because the overhead can be hidden by the overwhelming I/O time. Instead, we use user time and sys time, the time that measures the process running in user mode and kernel mode, correspondingly. These time gives us an accurate understanding of the overhead.

As shown in figure 3, when we monitor all functions, the user time increases by about 10%–20%, but the system time increases dramatically. We attribute the increase in user time to the flushing of cache and pipelines. In the GNU programs we measured 3-5 times overhead due to our waypoint mechanism.

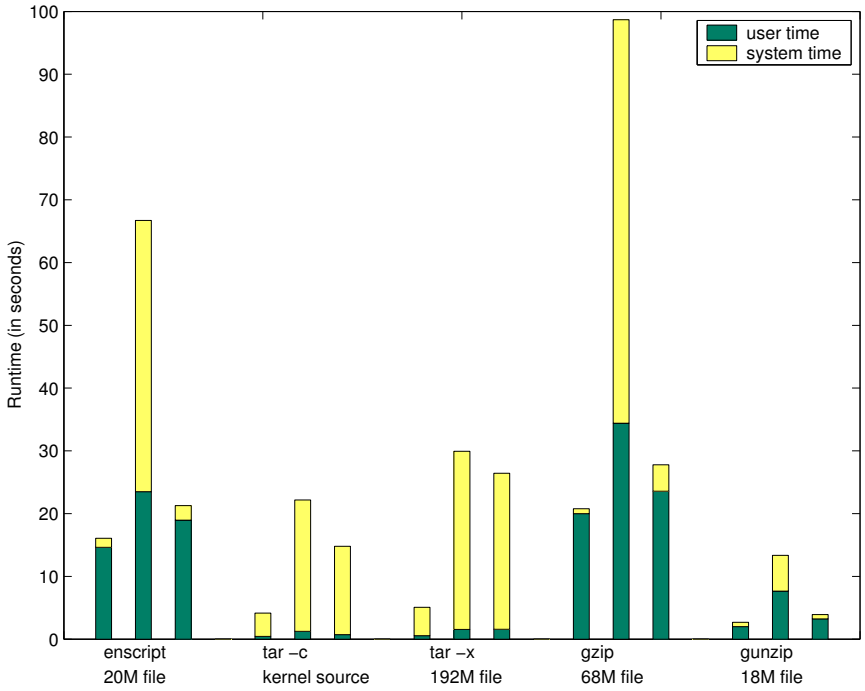

Fig. 3. overhead of the waypoint-based access monitor. For every group, the left side bar shows the time of running the original program; the middle bar shows its running time under waypoint-based access monitoring for all functions; and the right side bar shows the result with monitoring only functions that invoke dangerous system calls.

When we monitor only dangerous functions, the overhead is smaller than for monitoring all functions. Dangerous-function monitoring for `enscript`, `gzip` and `gunzip` introduces small overhead, but the overhead for `tar` is still high. In `gunzip`, there are only a few function calls for checking the zip file and for decompressing it. Because there are only a few waypoint invocations in the entire program execution, the running time is close to the original running time. We conclude that the overhead depends on not only how many functions are monitored, but also how frequently these functions are invoked.

5 Related Work

There are three layers of defense in preventing attacks from subverting the system. The first layer of defense is to prevent the malicious data and code from being injected, typically by avoiding and tolerating implementation errors in the system. Existing techniques include language-based or compiler-based techniques, such as type checking [9, 30–32], or protecting data pointers [33] and format strings [3]. The second layer of defense is to prevent malicious code from being executed. Prevention methods include instruction set randomization [34, 35], non-executable stack and heap pages [8, 10], process image randomization [10, 13], and stack integrity guarding [4, 11]. The third layer of defense attempts to prevent the executing attack code from doing further harm though the system interface. Existing work at this stage includes anomaly detection [5, 6, 12, 24, 25, 27], process randomization [2, 10, 13, 28, 36], and instruction set randomization [34, 35].

Realizing that lack of context information in detection leads to certain false negatives possible (e.g., the impossible-path problem and the mimicry attacks), some anomaly monitors apply partial context information in anomaly detection [5, 24, 25]. The benefit of using context information is that control path information between two system call invocations can help detecting anomaly.

Retrieving user call stack information in system call interceptor [5] is promising in bringing function call flow information to the anomaly monitor. We explore this approach further by providing trustworthy control flow information to the monitor. One other difference is that while [5] emphasize the call stack signature at a system call invocation, we put much effort on guarding with the permissions of application functions. Program shepherding [25] uses an interpretor to monitor the control flow of a process. It enforces application code to call library functions only through certain library entrance points, and the target library function must be one of the external functions listed in the external symbol table of the application executable. Because program shepherding does not monitor the data flow, some control flow information, such as function pointers, may be overwritten. If the overwritten pointer happens to be a library entry point, and the attack chooses a library function that is used at any other locations in the program, the attack can pass the check. Context related permissions can help in this situation. [24] associates a system call with its invocation address. The return-into-lib(c) attack calls a library function, rather than a piece of shellcode. In this case, the locations do not provide enough control path information to the detector.

6 Conclusion

In this paper, we propose a new mechanism—waypoints—to provide trustworthy control flow information for anomaly monitoring. We demonstrated how to use our waypoint mechanism to detect global mimicry attacks. Our approach can also catch return-into-others impossible paths by guarding the return addresses. Implementing waypoints by kernel traps provides reliable control path information, but slows down an ordinary program by 3-5 times. As a trade-off, by monitoring only dangerous system calls, we can reduce the overhead by 16%-70%, but no longer monitor the complete function call path.

As noted in our discussion of access to the raw system interface, waypoint-based detection cannot find local mimicry attacks, because the function has the proper permissions required to invoke the dangerous system calls. In our current implementation, we associate a permission set with each waypoint, but a state machine can provide tighter monitoring than a set. We will also investigate the use of complementary techniques, such as parameter checking, to extend waypoints to defend against local mimicry attacks.

Impossible paths may be generated at multiple granularities. Our waypoint mechanism can detect only function-granular return-into-others impossible paths by guarding return addresses.

Our waypoint mechanism cannot directly detect attacks through interpreted code. Because we work at the binary code level, our mechanism does not "see" the interpreted code. Rather, it monitors the interpreter itself, and so only sees actions taken by the interpreter in response to directives in the interpreted code.

So far, we generate waypoints and their permissions statically, which does not support self-loading code. Our future work will be to support self-loading code by moving waypoint set up procedure (by code instrumentation) to program load time. Additional future work is to optimize performance. Some optimizations that we have discussed are hoisting waypoints out of loops and merging waypoints for several consecutively called functions.

Our prototype implementation of the waypoint mechanism for Linux X86 system may be downloaded from http://www.sai.syr.edu/projects.

Acknowledgments

We thank Kyung-suk Lhee and the anonymous referees for their helpful comments.

References

1. Baratloo, A., Tsai, T., Singh, N.: Libsafe: Protecting critical elements of stacks. Technical report, Avaya Labs Research (1999)
2. Chew, M., Song, D.: Mitigating buffer overflows by operating system randomization. Technical report, CMU department of computer science (2002)
3. Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G., Frantzen, M., Lokier, J.: FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In: proceedings of the 2001 USENIX Security Symposium, Washington D.C. (2001)
4. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: Proceedings of the 7th USENIX Security Symposium, San Antonio, Texas (1998)
5. Feng, H.H., Kolesnikov, O.M., Fogla, P., Lee, W., Gong, W.: Anomaly Detection Using Call Stack Information. In: Proceedings of the 2003 IEEE Symposium on Security and Privacy, Berkeley, CA (2003)
6. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A Sense of Self for Unix Processes. In: Proceedings of the 1996 IEEE Symposium on Security and Privacy. (1996)
7. Purczynski, W.: (kNoX—implementation of non-executable page protection mechanism)

8. Solar Designer: Non-Executable User Stack, (http://www.openwall.com/linux/)
9. Lhee, K., Chapin, S.J.: Type-Assisted Dynamic Buffer Overflow Detection. In: Proceedings of the 11th USENIX Security Symposium, San Francisco (2002)
10. the Pax team: design & implementation of PaX, (http://pageexec.virtualave.net/docs/index.html)
11. Vendicator: StackShield: A "stack smashing" technique protection tool for linux, (http://www.angelfire.com/sk/stackshield/)
12. Wagner, D., Dean, D.: Intrusion detection via static analysis. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy. (2001)
13. Xu, J., Kalbarczyk, Z., Iyer, R.K.: Transparent Runtime Randomization for Security. In: Proceedings of the 22nd Symposium on Reliable and Distributed Systems (SRDS), Florence, Italy (2003)
14. Ghosh, A., Schwartzbard, A.: A study in using neural networks for anomaly and misuse detection. In: 8th USENIX security symposium. (1999)
15. Lee, W., Stolfo, S.: Data mining approaches for intrusion detection. In: 7th USENIX security symposium, San Antonio, TX (1998)
16. Warrender, C., Forrest, S., Pearlmutter, B.: Detecting Intrusions Using System Calls: Alternative Data Models. In: Proceedings of the 1999 IEEE Symposium on Security and Privacy. (1999)
17. Wespi, A., Dacier, M., Debar, H.: Intrusion detection using variable-length audit trail patterns. In: 3rd International workshop on the recent advances in intrusion detection. Volume LNCS 1907, Springer. (2000)
18. Abadi, M., Fournet, C.: Access control based on execution history. In: Proceedings of the 2003 Network and Distributed System Security Symposium. (2003)
19. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: Proceedings of the 9th ACM Conference On Computer And Communication Security, Washington, DC, USA (2002)
20. Aleph One: Smashing The Stack For Fun And Profit. www.Phrack.org **49** (1996)
21. Nergal: The advanced return-into-lib(c) exploits. www.Phrack.org **58** (2001)
22. Box, D.: Essential .NET, Volume I: The Common Language Runtime. Addison Wesley (2002)
23. Gong, L., Ellison, G., Dageforde, M.: Inside Java 2 Platform Security: Architecture, API Design, and Implementation (2nd Edition). Addison Wesley (1999)
24. Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A fast automaton-based method for detecting anomalous program behaviors. In: Proceedings of the IEEE Symposium on Security and Privacy, IEEE Computer Society (2001) 144
25. Kiriansky, V., Bruening, D., Amarasinghe, S.: Secure execution via program shepherding. In: Proceedings of the 11th USENIX Security Symposium, San Francisco, CA (2002)
26. Bernaschi, M., Gabrielli, E., Mancini, L.V.: Enhancements to the linux kernel for blocking buffer overflow based attacks. In: 4th Linux showcase & conference. (2000)
27. Sekar, R., Venkatakrishnan, V., Basu, S., Bhatkar, S., DuVarney, D.C.: Model-carrying code: a practical approach for safe execution of untrusted applications. In: Proceedings of the nineteenth ACM symposium on Operating systems principles, ACM Press (2003) 15–28
28. Somayaji, A., Hofmeyr, S., Forrest, S.: Principles of a Computer Immune System. In: Proceedings of the 1997 New Security Paradigms Workshop, UK (1997)
29. Red Hat security: Updated kon2 packages fix buffer overflow (2003)
30. Ashcraft, K., Engler, D.R.: Using programmer-written compiler extensions to catch security holes. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy, Oakland, CA (2002)
31. Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97), Paris (1997) 106–119

32. Lhee, K., Chapin, S.J.: Buffer Overflow and Format String Overflow Vulnerabilities. Software—Practice & Experience **33** (2003) 423–460
33. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: Pointguard: Protecting pointers from buffer overflow vulnerabilities. In: Proceedings of the 12th USENIX Security Symposium. (2003)
34. Barrantes, E.G., Ackley, D.H., Forrest, S., Palmer, T.S., Stefanovic, D., Zovi, D.D.: Randomized instruction set emulation to disrupt binary code injection attacks . In: Proceedings of the 10th ACM Conference On Computer And Communication Security. (2003)
35. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering Code-Injection Attacks With Instruction-Set Randomization. In: Proceedings of the 10th ACM Conference On Computer And Communication Security. (2003)
36. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In: Proceedings of the 12th USENIX Security Symposium, Washington D.C. (2003)