

To appear in *Proc. IFIP Eighteenth Int. Inf. Security Conf.*, Kluwer Acad. Publishers (Athens, Greece, May 2003).

## **An Experiment in Software Decoy Design**

### *Intrusion Detection and Countermeasures via System Call Instrumentation*

J. Bret Michael, Georgios Fragkos, and Mikhail Auguston

*Naval Postgraduate School, Department of Computer Science, Monterey, California USA*<sup>1</sup>

*Hellenic Army General Staff, Holargos, Athens, GREECE*<sup>2</sup>

**Abstract:** This paper presents an implementation of integrated intrusion detection and system response based on system call instrumentation. We introduce the notion of an intelligent software decoy as a means for detection and response to patterns of suspicious behavior. A prototype of such a system has been developed using NAI Labs' Generic Software Wrapper Toolkit. We present a case study of an ftp-based intrusion.

**Key words:** Automatic instrumentation, Deception, Intrusion detection, Intrusion response

## **1. INTRODUCTION**

Two types of strategies for defending against attacks in cyberspace are in wide use: identifying and fixing known vulnerabilities of an information system, and detecting attacks before they inflict significant damage on an information system or legitimate users of the system.

These strategies are not sufficient to ensure either the survivability or the intrusion tolerance of critical information systems, such as those comprising the information infrastructure of a nation state. These systems have to both survive and tolerate attacks perpetrated by highly trained aggressors, known as information warriors, who unlike script-kiddies, continually customize their existing arsenal of attack programs and create new ones in order to both avoid detection and achieve the maximum desired effect.

Michael *et al.* introduced in [7] a different approach to defending information systems, founded on the notion of *intelligent software decoys*, to counter the attacks of information warriors. The approach has both a pro-

tection and counterintelligence component. The decoy consists of one or more software wrappers placed around a unit of software (*e.g.*, component or method), with each wrapper consisting of a set of rules for detecting and responding to suspicious behavior. Instead of indicating to the attacker that he has been detected, the decoy keeps the attacker occupied by creating the illusion for the attacker that the attack is progressing as expected, using techniques ranging from fake error messages to redirecting the interaction with the attacking computer process to a virtual sandbox (*e.g.*, via “dazzlement” [2]). The goal is threefold: to gather information about the nature of the attack, adjust the system’s defenses based on the intelligence information, and cause the attacker to experience an opportunity cost (*e.g.*, waste attack resources that could have been better applied, or expose sources and methods).

There are two basic requirements for this approach to be successful: being able to detect the attack, and responding without human intervention. Michael *et al.* in [6] propose the use of an event-based language to meet these two requirements. This language uses event patterns to define suspicious behavior and the actions to be taken when the events occur.

In this paper, we describe the next step, which was to design a prototype system and then select an exemplar attack, use our high-level language to specify the decoys to be used to counter the attack, manually translate the high-level specifications into a representation that could then be automatically converted into executable kernel modules via NAI’s Generic Software Wrapper Toolkit [5], and test the decoys against the attack program.<sup>3</sup>

## 2. CHAMELEON SPECIFICATION LANGUAGE

Our high-level decoy specification language, called CHAMELEON and introduced by Michael *et al.* in [6], provides for defining detection-and-response actions based on computations over event traces; it encompasses the characteristics of a cross section of the six classes of attack languages identified by Vigna *et al.* [11].<sup>4</sup>

An *event* is any action that can be detected during program execution. An example of an event is a system call, such as `read`. Events can have attributes. For instance, the following statement declares that the `read` event has two attributes: `buf` (the buffer) and `nbyte` (the size of the buffer).

```
event read (buf, nbyte)
```

If we want to refer to one of these attributes we use the syntax `buf(read)`.

Two binary relations are defined for events, precedence and inclusion. An event may occur before another event or an event may be included inside

another event. For example, the following axiom specifies that an event of type `open_running_processes` contains an event of type `EnumProcesses` followed by a set of one or more unordered events of type `OpenProcess`.

```
open_running_processes::(EnumProcesses {OpenProcess}+)
```

These two relations suffice to describe a program execution as a partially ordered set of events, that is, an *event trace*. The set of all events during a program's execution is contained in an event called `execute-program`.

An expression containing events and conditions on their attributes is an event pattern. The following event pattern matches any `read` system call with a buffer size larger than 1000.

```
x: read & nbytes(x) > 1000
```

In the above example the name `x` is associated with the specific instance of the event `read`.

Events have duration, a beginning, and an end. We can select from a trace only those events that match a specific pattern using the keyword `detect`. For example, the following statement selects a `read` event that has a buffer size larger than 1000 from the set of events that occur during the program execution.

```
detect x: read & nbytes(x) > 1000 from execute-program
```

A `detect` statement can also contain a *probe*. A probe is a Boolean expression containing event attributes, subroutine calls, or a combination of the two and is executed immediately after the previous event pattern has been successfully matched. Probes can be used to specify additional conditions for filtering events. For example, the following expression specifies that a user other than `root` attempts to write to the password file.

```
x: write & filename(x) == '/etc/passwd'
  probe (user != 'root')
```

An event pattern combined with an action forms a rule. When an event that matches the event pattern is detected, the action is performed. The action part of the rule is specified with the keyword `do` and contains C-like statements. The following rule specifies that each time a `read` event is detected, and the buffer contains the string "SITE EXEC", then the value "NOOP" should be assigned to the buffer.

```
detect x: read & post (buf(x) == "SITE EXEC")
  from execute-program do buf(x) = "NOOP"
```

A sequence of decoy rules comprises a decoy specification; the sequence determines the order in which events will be detected and responses will be

generated. The complexity of the attacks and the intricacy of deception tactics make it impractical to develop decoy specifications using only a plain sequence of rules. The following statement specifies that first we detect `rule_1` and then either `rule_2` or `rule_3`.

```
rule_1 (rule_2 | rule_3)
```

In the statement shown below, we expect `rule_1` first, then one or more occurrences of the sequence (`rule_2 rule_3`), then `rule_4` might be detected, and finally `rule_5` might be detected zero or more times.

```
rule_1 (rule_2 rule_3)+ rule_4? rule_5*
```

In our approach, the decoy specifications are transformed into kernel modules. These modules run in kernel space and have unrestricted access to the entire operating system. The compilation of the decoy specifications produces wrapper definitions for the Generic Software Wrapper Toolkit, which in turn produces the kernel modules.

### 3. CASE STUDY

We demonstrate how we would instrument system calls to try to detect and respond to an attack. We start with a description of a variant of a well-known attack script, and then walk through the specification of actions we would like the software decoys to take to try to both detect and respond to the attack while simultaneously deceiving the attacker about the true nature of the effects of the attack on the decoy-enabled units of software.

#### 3.1 Attack

Certain versions of Washington University's ftp server (`wu-ftpd`) contain an input-validation vulnerability associated with the `SITE EXEC` command (*vid.* [13]), which if exploited, can give root privileges to a remote user. Due to the widespread use of `wu-ftpd`, many programs have been developed that automate the exploitation of this vulnerability. Information warriors would also create their own variants of `wu-ftpd` attack programs, for instance to improve the speed of execution of the attack or to make it difficult for the adversary to link the attack program back to its creator. One of these programs is `autowux.c` [1], which via a series of specially formatted `SITE EXEC` commands overwrites the return address on the stack and executes arbitrary commands as root. The attack itself consists of eight steps.

**Logging In (Step 1)**

The attacker logs anonymously into the ftp server. Anonymous ftp does not require a specific password from the user, so when asked for a password the attacker sends a special string called *shellcode* that is treated as a series of machine language instructions, which if executed by the ftp server, can spawn a shell. Commands executed through this shell will have root privileges. The two steps are to first store the shellcode somewhere in the memory of the targeted computer (the anonymous user's password is stored in the server's memory) and then try to force the target to execute the shellcode.

**Checking Vulnerability and Finding Buffer Address and EIP Location (Steps 2 - 6)**

Next, the command "SITE EXEC %.f" is sent to test whether the server is vulnerable; if the server is not vulnerable, the attack script terminates itself.

The attacker sends a series of specially formatted "SITE EXEC" commands to find the location in the stack where the shellcode and the execution instruction pointer (EIP) reside. When a program calls a subroutine, this address is stored in the stack. After the subroutine ends, the EIP value is fetched from the stack. By changing this value before it is fetched, the attacker can execute the instructions stored in the location pointed to by the new value.

**Exploiting and Starting the Shell (Steps 7 - 8)**

Based on the information collected in the previous steps, the attacker sends a "SITE EXEC" command that substitutes the value in the location where the EIP is stored with the address where the shellcode is stored; the shellcode will be executed spawning a shell with root privileges. If the attack is successful, the attacker interacts with the shell instead of the ftp server. To confirm success, the attacker sends the "id" command. This command should be executed giving back the expected result.

At this stage the attack is considered to be complete. The attack program enters an infinite loop sending the user input to the server and handling the responses.

**3.2 Foiling the attack via deception**

We applied a simple deception strategy for the purpose of demonstrating our approach to specification and instrumentation: make the attacker believe that the attack proceeds as expected, while simultaneously protecting the server from executing any dangerous commands. We first specify, using the CHAMELEON language, the detection and response actions to be performed

by the software decoy. We then demonstrate how these specifications would be mapped from the high-level specification to the equivalent representation that the NAI Generic Software Wrapper Toolkit needs to generate the wrappers around system calls (*i.e.*, kernel modules) for the underlying operating system; we intend to implement a compiler to automate the translation process.

Since this is a remote attack, the targeted system only has access to the network traffic exchanged between the ftp server and the attacker: commands and responses. The only means by which a decoy can intervene during the attack is by intercepting and modifying this traffic. The ftp server communicates with the attacker with the help of two system calls: `read` and `write`. The wrappers can give access to these system calls and their parameters. The decoy can intercept these two calls and change the contents of the buffer passed as a parameter, substituting simulated commands and faked responses. The decoy must both substitute the commands before they reach the ftp server and the responses before they are transmitted over the network. One of the wrapper's features is the ability to intercept system calls either just before they are executed or after execution is complete, indicated by the keywords `pre` and `post`, respectively.

As mentioned above, the attack starts with the shellcode being sent as the password. We assume that each time the decoy detects the shellcode during a read operation, it needs to treat the corresponding slice of the event trace as an attack; it is unlikely that a benign user would submit a shellcode as a password. If the event slice is not detected, then the decoy does not proceed to the next step. The following is the wrapper definition code.

```
/* STEP 1 */
op{read} && step (((char *)$iobuf) =~ m|shellCode| )
  post {wr_printf("FTP Wrapper: STEP 1.\n");
        attackStep = 1; };
```

In the second step, the attacker sends the command "SITE EXEC `%.f`" to test if the ftp server is vulnerable to the attack. When executed, this command returns two lines of response, which the decoy must emulate. The decoy must also modify the value of `$sizeret`, which is the value that `read` returns, indicating the number of characters read: this value must include an extra newline character at the end.

```
/* STEP 2a */
op{read}&& step((((char *)$iobuf)=~
  m|^SITE EXEC %.f| )) post {char * newbuf;
  wr_printf("FTP Wrapper: STEP 2\n");
  attackStep = 2;
  /* Replace SITE EXEC command w/ a harmless command */
```

```

/* that causes the server to respond w/ two lines */
newbuf = wr_strdup("NOOP\n\n");
delete $iobuf; $iobuf = newbuf;
$sizeret = 6; /* Change the return value of the */
             /* read system call to reflect the */
             /* changes we made to the buffer */ };

```

Once the decoy has intercepted the read system call, and substituted the buffer, the decoy must intercept the response from the ftp server and modify it before it reaches the attacker. The substitution must be done in such a way that the attack program receives exactly what it expects. The decoy rules we created to do this are shown in steps 2b and 2c.

```

/* STEP 2b */
op{write} && step (((char *)$iobuf) =~ m|^200|)
           && (attackStep == 2)) pre {char * newbuf;
/* Replace the error message with what the attacker
   expects. */
newbuf = wr_strdup("200--2\r\n");
delete $iobuf; $iobuf = newbuf; $sizeret = 8; };

/* STEP 2c */
op{write} && step (((char *)$iobuf) =~ m|^500|)
           && (attackStep == 2)) pre { char * newbuf;
/* Replace the error message with what the attacker
   expects. */
newbuf = wr_strdup("200 (end of '%.f')\r\n");
delete $iobuf; $iobuf = newbuf; $sizeret = 21; };

```

Steps 3 to 5 work in a way similar to step 2. The decoy intercepts the ftp commands before they reach the ftp server and substitute them. Likewise, the decoy intercepts the response from the ftp server and substitutes it with what the attack program expects. This way, the ftp server never executes any commands that could compromise it, and the attack program is deceived into believing that the attack proceeds as expected.

Step 6 poses a challenge, because, during this phase of the attack, the server is expected to crash several times. One way of simulating the crash is to make the ftp server close the connection. If during a read operation the decoy changes the value of `$sizeret` to zero, the ftp server will determine that it has reached the end of file (EOF) and close the connection.

```

/* STEP 6a */
case op{read} && step (((char *)$iobuf) =~ m|^SITE EXEC
%| ) post { char * newbuf;
  if (((char *)$iobuf) =~ m|^SITE EXEC %3093$x|)

```

```

|| (((char *)$iobuf) =~ m|^SITE EXEC %3094$x| )
|| (((char *)$iobuf) =~ m|^SITE EXEC %3127$x| )
|| (((char *)$iobuf) =~ m|^SITE EXEC %3144$x| )
|| (((char *)$iobuf) =~ m|^SITE EXEC %3145$x| )
|| (((char *)$iobuf) =~ m|^SITE EXEC %3150$x| )
|| (((char *)$iobuf) =~ m|^SITE EXEC %3151$x| )
|| (((char *)$iobuf) =~ m|^SITE EXEC %3152$x| ) ){
    $sizeret = 0; /* EOF */
} else {
/* Replace SITE EXEC command with a harmless command */
    newbuf = wr_strdup("NOOP\n\n");
    delete $iobuf; $iobuf = newbuf; $sizeret = 6; } };

```

### 3.3 High-level specification

The high-level specification follows exactly the sequence of steps of the low-level wrapper specification. The most important parts of the specification are discussed here; the complete specification is given in [4].

We declare two events: `read` and `write`.

```

event read(iobuf, sizeret)
event write(iobuf, sizeret, nytes)

```

Next, we specify a rule for each step in the wrapper definition. Step 1 is a rule with no action part. It only has a `detect` part. When the specified event pattern is detected, the decoy proceeds to the next step waiting for the next event to be detected.

```

step_1:: detect x: read & post (iobuf(x) == shellcode)
    from execute-program

```

Step 2 contains three rules. The logic in these rules is the same as in the wrapper definition.

```

step_2a:: detect x: read
    & post (iobuf(x) == "SITE EXEC %.f")
    from execute-program
    do {iobuf(x) = "NOOP\n\n"
        sizeret(x) = 6 }

step_2b:: detect x: write & pre (iobuf(x) == "^200")
    from execute-program
    do {iobuf(x) = "200--2\r\n"
        sizeret(x) = 8 }

step_2c:: detect x: write & pre (iobuf(x) == "^500")

```

```

from execute-program
do {iobuf(x) = "200 (end of '%.f')\r\n"
    sizeret(x) = 21 }

```

Steps 3 through 8 are specified in a similar manner. The part of the decoy specification that does the actual work is the last one, where we specify a composite rule, named `ftp_decoy`, consisting of all the previous rules.

```

ftp_decoy::
step_1 step_2a step_2b step_2c
(step_3_4_a step_3_4_b step_3_4_c)*
step_5 (step_5a step_5b step_5c)*
step_6_I (step_6a_I step_6b_I step_6c_I)*
step_6_7_II (step_6a_II step_6b_II step_6c_II)*
step_8a step_8b step_8c step_8d

```

Although this wrapper deceives the attacker into believing that the attack was successful, the deception ends when the attacker tries to interact with the shell. The shell functionality is not simulated and so the attacker will discover that something went wrong and possibly suspect that the targeted ftp server utilizes a deception mechanism. There are solutions to this problem. For instance, a library could provide all of the shell functionality, and be used to maintain the deception. Alternatively, the decoy could carry on the deception by transferring the attacker to another virtual machine where everything is simulated (*e.g.*, an *ante chamber* as described in [7]).

We submitted the low-level specification to the Generic Software Wrapper Toolkit, which produced a decoy for use with the Linux operating system. We then ran the attack script against the decoy-enabled ftp server; the decoy handled all of the system calls, as expected.

## 4. RELATED WORK

Sekar *et al.* [8] and Vankamamidi [10] developed the high-level Auditing Specification Language (ASL), with the aim of making information systems survivable. Their goal was to make systems capable of operating and offering their services even in the presence of vulnerabilities. This is achieved by detecting attacks in real-time and isolating them before they can cause significant levels of damage.

A program's intended behavior is described in ASL as a set of assertions. Any behavior not conforming to these assertions is treated as an intrusion. A process' behavior is observed through the system calls that it makes. An ASL specification involves a series of rules. Each rule consists of an event

pattern and an action. The ASL specifications are compiled into C++ classes, which are then used to generate detection engines.

Eckmann *et al.* in [3] designed the STATL language to be extensible; it can be expanded via extension modules that contain domain-specific types, variables, and events. STATL supports the State Transition Analysis Technique [12] for detecting intrusions. Attack scenarios are described as a series of states and transitions. Each transition has an action associated with it. In order to abstract away the details of the modeled attacks, only the events without which the attack would fail are used. Further abstraction is achieved by describing the actions using higher-level representations, so that actions with the same effect, but different implementations, can have the same representation. The attack scenarios are compiled into dynamically linked modules called “scenario plugins.”

Ko *et al.* [5] used the Generic Software Wrappers Toolkit to integrate intrusion detection and response functions into the kernels of operating systems. They treat software wrappers as state machines that intercept system calls. Wrappers are defined using the Wrapper Definition Language (WDL). The language describes the events that are going to be intercepted and the actions that the wrapper will take when these events are detected. The WDL specifications are compiled with the help of the Wrapper Compiler (WrapC) into native code.

The Wrapper Support Subsystem (WSS) facilitates the configuration and management of the wrappers. The wrapper modules are inserted into the kernel dynamically. Once a wrapper is loaded into the kernel it can wrap any process according to activation criteria defined by the administrator. Wrappers can be layered. Additionally, more than one wrapper can be associated with a program at the same time, each one implementing a different detection technique.

Finally, Templeton *et al.* in [9] argues that current intrusion detection techniques fail to detect multi-staged attacks, unknown attacks, or variations of known attacks. Their approach treats attacks as a set of capabilities. An attack is described as a set of abstract attack concepts. Their language, called JIGSAW, can be used to model the abstract concepts. Their approach can be used to discover new attacks or coordinated attacks across many systems.

## 5. CONCLUSION

The notion of intelligent software decoys has only begun to be explored. Nevertheless, their potential value, in protecting critical information systems, is apparent. The work presented in this paper addresses one aspect of decoy

technology: the automatic instrumentation of software with detection-and-response capabilities.

New constructs and new concepts were added to the CHAMELEON language, improving its richness and expressiveness. The case study serves as the basis for demonstrating the expressiveness of the language for specifying decoy actions. The results of this experiment were satisfactory, since the kernel modules respond to the attack program as specified.

The Generic Software Wrapper Toolkit proved to be a valuable tool for controlling the interaction, in the form of system calls, between the `autowux.c` program and the decoy-enabled ftp server. The problem is that the wrapper definitions are low level, and implementing them requires knowledge about system calls and their parameters. The CHAMELEON language reduces the complexity of the definitions, making it easier for people to write and understand these definitions. We are currently working through more case studies to identify additional language features before beginning work on building a compiler that will automatically create wrapper definitions (in NAI's Wrapper Definition Language) from specifications written in the CHAMELEON language.

## NOTES

1. Profs. Michael and Auguston can be contacted at {bmichael | maugusto}@nps.navy.mil
2. CPT Fragkos can be reached at gfra@softhome.net
3. Conducted under the auspices of the Naval Postgraduate School's Homeland Security Leadership Development Program, this research is supported by the U.S. Department of Justice Office of Justice Programs and Office for Domestic Preparedness. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotations thereon.
4. We chose the name "Chameleon" because, just as a the lizard of the genus *Chamaeleo* can change both its color and behavior to deceive its predators and catch prey, our high-level language is meant to be used to formally specify how an intelligent software decoy is to adapt both its signature (*e.g.*, via the use of polymorphic argument types as described in [7]) and actions (*e.g.*, pretend to be vulnerable) to deceive and catch attackers.

## REFERENCES

- [1] Anonymous. Software program named `autowux.c` - wu-ftpd remote root exploit for x86/linux up to version 2.6.0, dated May 4, 2001.

- [2] Cohen, F. and Koike, D. Leading attackers through attack graphs with deceptions. Unpublished manuscript, Sandia National Laboratories, Livermore, Calif., May 29, 2002.
- [3] Eckmann, S. T., Vigna, G., Kemmerer, R. A. STATL: An attack language for state-based intrusion detection. *J. Computer Security* 10, 1/2 (2002), 71-104.
- [4] Fragkos, G. An event-trace language for software decoys. M.S. thesis, Dept. of Computer Science, Naval Postgraduate School, Sept. 2002.
- [5] Ko, C., Fraser, T., Badger, L., Kilpatrick, D. Detecting and countering system intrusions using software wrappers. In *Proc. Ninth USENIX Sec. Symp.*, USENIX Assn. (Denver, Colo., Aug. 2000), 145-156.
- [6] Michael, J. B., Auguston, M., Rowe, N. C., and Riehle, R. D. Software decoys: Intrusion detection and countermeasures. In *Proc. Workshop on Inf. Assurance*, IEEE (West Point, N.Y., June 2002), 130-138.
- [7] Michael, J. B. and Riehle, R. D. Intelligent software decoys. In *Proc. Monterey Workshop: Eng. Automation for Software Intensive Syst. Integration*, Naval Postgraduate School (Monterey, Calif., June 2001), 178-187.
- [8] Sekar, R., Bowen, T., and Segal, M. On preventing intrusions by process behavior monitoring. In *Proc. Workshop on Intrusion Detection and Network Monitoring*, USENIX Assn. (Santa Clara, Calif., April 1999), 29-40.
- [9] Templeton, S.J., Levitt, K. A requires/provides model for computer attacks. In *Proc. New Security Paradigms Workshop*, ACM (Cork, Ireland, Sept. 2000), 31-38.
- [10] Vankamamidi, R. ASL: A specification language for intrusion detection and network monitoring. M.S. thesis, Dept. of Computer Science, Iowa State University, Ames, Iowa, Dec. 1998.
- [11] Vigna, G., Eckmann, S. T., Kemmerer, R. A. Attack languages. In *Proc. Inf. Survivability Workshop*, IEEE (Boston, Mass., Oct. 2000).
- [12] Vigna, G., Eckmann, S. T., Kemmerer, R. A. The STAT tool suite. In *Proc. DARPA Inf. Survivability Conf. and Exposition*, vol. 2, IEEE (Hilton Head, S.C., Jan. 2000), 46-55.
- [13] Widespread exploitation of rpc.statd and wu-ftpd vulnerabilities. Incident note IN-2000-10, CERT Coordination Center, Carnegie Mellon University, Pittsburgh, Penn., Sept. 15, 2000.