Contract-Based Security Monitors for Service Oriented Software Architecture

Alexander M. Hoole
Dept. of Electrical and Computer Engineering
University of Victoria, BC, Canada
alex.hoole@ece.uvic.ca

Issa Traore
Dept. of Electrical and Computer Engineering
University of Victoria, BC, Canada
itraore@ece.uvic.ca

Abstract

Monitors have been used for real-time systems to ensure proper behavior; however, most approaches do not allow for the addition of relevant fields required to identify and react to security vulnerabilities. Contracts can provide a useful mechanism for identifying and tracking vulnerabilities. Currently, contracts have been proposed for reliability and formal verification; yet, their use in security is limited. Static analysis methods are able to identify many known vulnerabilities; however, they suffer from a high rate of false-positives. The creation of a mechanism that can verify identified vulnerabilities is therefore warranted. We propose a contract-based security assertion monitoring framework (CB_SAMF) for reducing the number of security vulnerabilities that are exploitable. CB_SAMF will span multiple software layers and be used in an enhanced systems development life cycle (SDLC) including service-oriented analysis and design (SOAD).

Keywords: contracts, monitors, security engineering, service-oriented architecture

1 Introduction

Recent years have seen widespread application of technologies, such as firewall and intrusion detection systems (IDS), intended to stem the damage caused to consumer systems. These technologies will never completely fix the security problem because they do not remove software containing security defects from systems. IDSs have been implemented as monitoring frameworks; yet, IDSs do not fix software vulnerabilities, they only track and potentially prevent them from being exploited [3, 6, 7]. Specifically, we still require better tools and methodologies to identify, reduce, and remove security defects in software systems.

Since service oriented architecture (SOA) often couples legacy systems with new systems, in a reconfigured services-oriented approach focusing on business process, security for such systems can be complex. While it is desirable to have security features designed as a service layer in the architecture, it does not guarantee security of the overall system. Especially when legacy systems are consumed by the services. Monitors can provide a useful tool during SOAD to determine if and when a particular feature or vulnerability is exploited. The use of contracts to specify monitoring probes allows for a looser coupling of security monitoring from actual code and a tighter coupling to security policy within a business process.

We propose a contract-based security assertion monitoring framework (CB_SAMF) for reducing the number of exploitable security vulnerabilities during design and testing. CB_SAMF is intended to detect events that contradict the specified acceptable use of a system. Acceptable use of a system, network, or application is usually specified in the form of a security policy document. Risk assessment and requirements gathering also contribute to the identification of security assertions. Ultimately, requirement documents specify the security requirements of the system.

Our form of contract, derived from requirement assertions, identifies environmental and system security conditions necessary for generation of probes which monitor security assertions during runtime. When a contract violation occurs during runtime, security vulnerabilities are identified and reactive countermeasures can be deployed.

Our long term objective in this work is to create artifacts that will allow for the transition of security requirements/goals throughout the SDLC by creating a model, framework, and set of tools to assist developers produce more secure and reliable services. In particular, we desire the removal of security vulnerabilities during design and testing rather than dependence on maintenance tools such as firewalls and IDSs.

The remainder of the paper is organized as follows: Section 2 discusses related work on monitors and contracts; Section 3 introduces our monitoring framework; Section 4 covers a small case study; and Section 5 presents our closing remarks.

978-0-7695-3473-2/08 $25.00 © 2008 IEEE
DOI 10.1109/APSCC.2008.169

2 Related Work

2.1 On Monitors

For security systems, several monitoring approaches were presented based on policy driven models [3, 4, 7, 15, 13]. For instance, [7] uses a specification based IDS approach focusing on utilizing security specifications to describe the intended behavior of programs. Then, during run-time, they produce traces of the monitored programs with the ultimate goal of performing real-time intrusion detection.

In [12], Peters and Parnas introduce requirements-based monitors, derived from the specification of the system, which are used to ensure that real-time systems behave correctly. Due to device limitations, this monitoring approach is prone to both false-positives and false-negatives.

The approach in [14] is also centered on real-time system monitoring and debugging practices. The monitoring framework, Ferret, is based on-top of a para-virtualized version of Linux. Ferret enables insertion of monitoring sensors using a sensor directory, registered monitors, and registered clients resulting in monitoring of a real-time system.

A more recent addition by Barringer *et al* is the EAGLE framework [2]. EAGLE's rule-based framework for implementing trace-monitoring logics is similar to the notion of contracts. EAGLE can represent future and past-time temporal logic, extended regular expressions, real-time and data constraints, state machines, interval logics, statistics, and forms of quantified temporal logics. While their approach focuses on verification of systems during runtime, it does not focus on security, collect information relating to security, nor provide a mechanism for reactive measures.

2.2 On Contracts

Using the notion of a contract to improve different design processes in software engineering has been proposed and researched by many individuals [5, 10, 8, 11].

Older work on design by contract for improving reliability in [11] is also an inspiration for this work. Meyer discusses preconditions, postconditions, and invariants as primary components in contracts toward improving reliability in software systems. A typical contract outlines benefits for each party and specific obligations. Meyer argued that through the use of "design by contract", developers could improve correctness and robustness of systems leading to the absence of bugs.

Contracts were applied as an approach for specifying concurrent systems in [1, 8]. Lamport's work provided a foundation for formal validation of systems based on their specifications and state transitions. With the ability to handle linear temporal logic statements, safety properties, and liveness properties there were multiple similarities to [2].

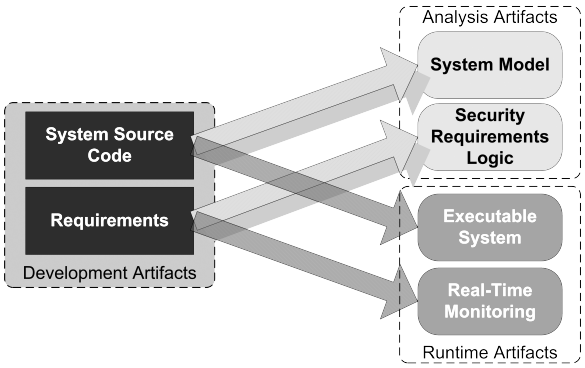

Figure 1. Relationship between analysis, development and runtime artifacts.

3 Monitoring Framework

We present in this section the key components of our proposed contract-based monitoring framework which relate analysis, development, and runtime artifacts. A possible notation for representing requirements is in the form of sequence diagrams and propositional temporal logic which can be directly related to the source code and runtime implementation as seen in Figure 1.

3.1 Security Contracts

The notion of a contract in software engineering is not a new idea [5, 10, 8, 11]. When applied to security, we must analyze carefully which properties need to be specified in a contract to improve security. Many pre and postconditions are more to do with robustness than security. Historically, preconditions specify when it is appropriate to call a particular feature (function/method), while postconditions specify what is true after a particular feature is called (what has been accomplished by the function/method).

Our contract needs to bind the caller and callee to deal with additional properties involving timing, property values, and other events. A **binding contract** is the legal agreement between two or more entities to perform and/or not perform a set of actions. For example, a contract specified for supplier *X*, consumed by consumer *Y*, guarantees that the supplier *X* has fulfilled the postcondition(s), provided that the consumer *Y* has satisfied the precondition(s). The consumer is protected from the supplier since the postconditions have been guaranteed by the supplier. The supplier is protected from the consumer since the preconditions have been guaranteed by the consumer. Our security contract model looks beyond (the traditional one) by including

the following fields to protect the supplier and consumer:

- Requirements - as preconditions (PRE)
- Guarantees - as postconditions (POST)
- References - as invariants (INV)
- Context - as relevant environmental information (CONT)
- History - as some knowledge keeping construct (HIST)
- Response - as a reactive measure (RESP)

The notion of contracts, as in [11], is not suitable for security monitoring. For example, under normal contracts, a false precondition does not guarantee that the system will not process the input. It may allow certain types of attacks, such as buffer overflows, to continue. The require, guarantee, and references fields of the contract, corresponding to the pre, post, and invariants mentioned above, do not handle all necessary attributes of security defects. We propose three new contractual fields. *Context* which provides support for environmental influences. *History* which provides support for complex vulnerabilities that sometimes result from a series of actions which may occur in parallel (both context and history can be useful when dealing with vulnerabilities such as DoS and race-conditions). Finally, since security assertion failures should be handled, not just detected (as in the case of contract proposed by Meyer), *response* provides a mechanism to choose how a particular assertion is handled when exploitation is detected. Resumption and organized panic for exception handling, used by Meyer, fit under our broader response category [11].

Work done in [2] relating to program monitoring and rule-based runtime verification has exposed interesting results. Specifically, work on linear temporal logic (LTL) and program states has been core to several attempts towards runtime verification and is a promising candidate for the notation of our contracts. As such, our notation is derived from LTL and is inspired by the EAGLE framework [2].

Each contract (*C*) contains a breakpoint (*B*) and one or more assertions (*A*). Breakpoints identify a monitoring location/symbol in the target application. The assertion is a rule which must remain true at the breakpoint. Each assertion has associated with it zero or more security contract extensions (*E*). Assertions can take on one of the following three forms: *PRE*, *POST*, or *INV*. We do not represent assertions types separately since they all take the same form. Each assertion is composed of zero or more rules (*R*), relating to the target (recall breakpoint *B*), and zero or more monitors (*M*). The rules, monitors, and extensions are individually named (*N*). Rules specify a property of the state of the program which needs to remain true, while a monitor enforces one or more rules. The quantifiers *min* and *max* represent liveness and safety properties respectively and are important for the boundary cases of a monitor trace [12]. Safety properties state that if a behavior is unacceptable any extension of that behavior is also unacceptable. Liveness properties state that for a given requirement, and any finite duration, the behavior can always be extended such that it satisfies the requirement[9, 12]. The body of every rule and monitor is specified as a boolean valued formula of the syntactic category *Form*. Finally, each contract may be instantiated with the following grammar expressed in EBNF:

$C := B\ (A\{E\})\ \{A\{E\}\};$
$E := \{CONT\} \mid \{HIST\} \mid \{RESP\};$
$A := \{R\}\{M\};$
$R := \{\underline{\text{max}}|\underline{\text{min}}\}\ N(T_1 x_1, ..., T_n x_n) = F;$
$M := \underline{\text{mon}}\ N = F;$
$T := \underline{\text{Form}} \mid \textit{primitive type};$
$B := \textit{symbol} \mid \textit{HEX address};$
$F := exp|\underline{\text{true}}|\underline{\text{false}}|\neg F|F_1 \wedge F_2|F_1 \vee F_2|F_1 \rightarrow F_2|\bigcirc F|\ominus F|$
$F_1 \cdot F_2|N(F_1, \ldots, F_n)|x_i;$
$CONT := \underline{env}\ N \mid \underline{res}\ N;$
$HIST := \underline{trace}\ N \mid \underline{runningsum}\ N \mid \underline{runningavg}\ N;$
$RESP := \underline{core}\ N \mid \underline{term}\ N \mid \underline{kill}\ N \mid \underline{log}\ N;$

We have also defined possible extended behaviors for context, history and response elements and may extend these in the future. Context may specify environmental or resource information (external to the program) which is needed by the contract. History may contain trace data or statistically relevant information. Finally, response may specify an action to perform when an assertion is violated.

With this definition it is possible to use multiple separate monitors or a single monitor.

3.2 Implementation

3.2.1 Overview.

We choose to implement our prototype monitoring framework on the Linux platform. During implementation we will be able to analyze existing code, perhaps using static analysis and other techniques, and combine this with the misuse case and attack tree derived information to create contracts intended to detect the exploitation of identified potential vulnerabilities. From the contracts we then generate, using a custom compiler, assertion-based probes for insertion during runtime to monitor executables. The probes will then feed information back into the controlling monitor framework for further analysis.

To implement such a system we require a mechanism to integrate the monitoring probes into existing systems. Some assertion based frameworks depend on modification of the original source code and compile time options [11]. Modification to the original binary can result in different runtime behavior than the unmodified version (as in overflows). We

desire a mechanism with minimal impact on the normal behavior of the system. One approach, which avoids the above issues, is to use software breakpoints. Breakpoints still effect the timing of an application; however, other approaches to monitoring also affect timing. The degree to which timing is affected depends on the work done at the breakpoint.

3.2.2 Breakpoints.

A breakpoint, in the context of software development, is typically a stopping or pausing point during software execution when a developer can inspect the current context of execution. As such, it provides a natural avenue for the development of a security monitoring and testing framework. Ironically, debuggers and breakpoints are also often the tools of malicious software hackers who are trying to break software security.

3.2.3 Probes.

A probe is an instrument or mechanism used to investigate and discover properties of something that is unknown. Probes are used throughout science and engineering for tasks such as space exploration, medical exams, failure detection and diagnosis, and software debugging. We propose to use a form of software probes to attach new functionality to existing code to analyze, diagnose and respond to security vulnerabilities.

In recent years kprobes was proposed and integrated into the Linux kernel. The kprobes Linux kernel patch, contributed by the Linux Technology Center (LTC) at IBM, added an extension to the normal software breakpoint mechanism allowing integration of new code into the kernel during runtime. The framework enables developers to add instances of the following three mechanisms to the kernel as kernel loadable modules:

1. Kprobes: integrate new functionality before or after an executable statement, or add fault handling code in the case where a fault arises during execution
2. Jprobes: access arguments passed to an executable statement, providing the ability to examine or override the default functionality of the statement
3. Returnprobes: examine or override the return value of an executable statement

4 Case Study

In this section, we illustrate the proposed monitoring framework by presenting a case study based on a buffer overflow vulnerability.

4.1 Buffer Overflow

Any function that does not perform bounds checking internally, and is "depending" on the caller to do the necessary checking, is unsafe in languages like C and C++ and could create the opportunity for a buffer overflow attack. To monitor such a condition we need to know the following:

- Target function for breakpoint
- Name or size of buffer argument being passed to function
- Size of the target buffer

Letting *x* represent the size of the source buffer and *y* the size of the destination buffer, a general contract statement can be generated. Essentially, we want to say "whenever we reach a state where $x = k$, for some k > 0, then eventually we reach a state where y == k, where the initial precondition that x must be less than or equal to y must hold".[1] Provided this assertion holds, we do not have an overflow.

4.2 Contract Model

If the above assertion is violated then we have shown that the vulnerability is exploitable. In linear temporal logic using first-order quantification, we can write the following:

$$\Box(x > 0 \rightarrow \exists\, k.(k = x \wedge \Diamond y = k)) \wedge \Box(x <= y)$$

The associated overflow contract takes the following form:

$$E = \underline{\log}\; buffer\_log$$
$$\underline{\min}\; R(int\; k) = Sometime(y == k)$$
$$\underline{\mathrm{mon}}\; M = Always(x > 0 \rightarrow R(x) \wedge x \leq y)$$
$$C = strcpy\; M\; E$$

In the above contract, we satisfy the LTL statement by creating a contract for the symbol *strcpy* which has one associated monitor (M) and one associated extension (E). The monitor observes two properties of the target breakpoint. First, whenever $x > 0$ implies that *y* will eventually equal *x*. This shows the necessary observation that when a source buffer is placed in a destination buffer, without checking the bounds, then our destination buffer will ultimately have the same length as the source (even though this is not valid for fixed length destination buffers). Second, to capture the illegal quality that the length of the source buffer cannot exceed the length of the destination buffer we have the boolean expression that *x* must be less-than or equal-to the *y*. If either of these assertions are evaluated as false we will have violated the contract and the resulting extension will be executed. In this situation, we show that a log entry is sent to *buffer_log* to track the event.

[1] The pseudocode uses '=' to refer to assignment, and '==' to refer to equivalence comparison. The contract uses a notation derived from Eagle for assignment and equivalence.

While contracts can be written for user-space functions, contracts can ultimately be created for any layer in the software system. Later we will show an example implementation in the kernel for a buffer-overflow problem.

4.3 Implementation Using kprobes

The monolithic kernel, containing the majority of kernel code, is the most sensitive environment in the Linux OS (containing the monolithic kernel and modules). Modules are used typically to implement device drivers and add additional functionality to the kernel during runtime but can only be inserted with root privileges. Below we explore a module to show that a buffer overflow occurring in the kernel is a severe security risk.

The first module provides a new file descriptor in the proc filesystem. The proc filesystem is a virtual filesystem providing an interface to exchange information between kernel and user-space. Two hooks are placed on the new file descriptor(/proc/target) for reading (read_target()) and writing (write_target()) operations. Once the module has been loaded and initialized, any read operation performed against the file descriptor will result in a call to *read_target* and any write operation will result in a call to *write_target*.

The following code is the main body of *write_target*:

```
/* Expect integer <= nine digits followed by a '\n' */
static int write_target(struct file *file,
const char *buffer, unsigned long count, void *data) {
   /* This code should be here to be secure */
   //if (count > sizeof(bad_string))
   // return -EINVAL;
   if (copy_from_user(bad_string, buffer, count))
      return -EFAULT;
   return count;
}
```

The two lines which are commented out are the root cause of the security vulnerability. The array variable *bad_string* was statically declared as a global variable of length 27. Since **buffer* is not being checked to ensure that its length will fit inside of the statically allocated space for *bad_string*, a buffer overflow is highly possible.

Reading the file descriptor will not cause any problems. Alternatively, a write operation invokes the write_target function with the potential for a buffer overflow. If a call is made to write containing a buffer with greater than 26 characters we will have an issue.

```
# echo -n "012345678901234567890123456789\
  012345678901234567890123456789" > /proc/target
```

When the example above executes, the system could enter a hung state (a form of denial of service caused by error propagation). The system could continue for a period of time before exhibiting some abnormal behavior. If a buffer overflow were to effect the call stack of a user space application an elevation of privilege attack could be mounted. The result is dependent upon the contents of memory that are overwritten and the code that uses that memory region. Regardless of the outcome, this behavior should not be allowed.

The redirection of standard out for the echo command results in a write operation against the file */proc/target*, passing the output of the echo argument as a parameter. As with most POSIX functions in Linux, the write function is implemented in the standard LIBC library to fire off a system call which then transfers control to the kernel to complete the operation. On the Intel architecture Linux uses the 0x80 software interrupt to transfer control to the kernel, at which point we enter the call chain indicated by Figure 2.

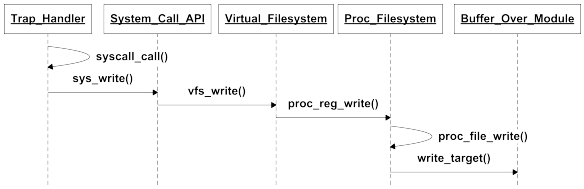

Figure 2. Stack trace of call to write_target from the kernel perspective.

For vulnerable kernel code we can make use of probes as the basis for our monitoring framework. The example probe has been manually written to show the feasibility of using probes as a monitoring mechanism. This example is similar to the example contract above. Noticeable differences include the name of the function that we are probing (the breakpoint), the writing to the standard ring-buffer (via *printk*) rather than a named log, and the focus on the length property only.

The kprobe mechanism needs the address of an executable instruction at which we wish to break. We obtain the address using the virtual file */proc/kallsyms* which exposes all kernel symbols with their related virtual addresses. Our probe also requires the address of the buffer on which the vulnerability is based. Thus, we load the modules as follows:

```
# insmod buffer_over.ko
# grep write_target /proc/kallsyms
e0930000 t write_target [buffer_over]
# grep bad_string /proc/kallsyms
e09305e4 d bad_string [buffer_over]
```

The first command loads the module into the kernel and initializes it. The second command queries the symbol table of the kernel to find the address of the function *write_target*. The first column shows the address of the symbol, the second column contains a 't' indicating that the symbol is located in the TEXT segment, the third column has the symbol name, and finally the buffer_over indicates that the symbol belongs to the named module. The final command

queries the kernel symbol table for the address of the buffer named *bad_string*. Observe that the second column has a 'd' indicating that the symbol is from the DATA segment. Then our probe is loaded into the kernel using the following command:

```
# insmod catch_buffer_probe.ko breakpoint=0xe0930000 \
  buffer_addr=0xe09305e4
```

When the module is loaded the two parameters shown above are brought into the module using the module_param macro and placed into the associated character strings.

```
char *breakpoint; /*parameter for breakpoint*/
char *buffer_addr; /*parameter for buffer*/
module_param(breakpoint, charp, 0400);
module_param(buffer_addr, charp, 0400);
```

To convert the addresses from character strings the probe defines variables for the breakpoint address, the vulnerable buffer, the buffer address, and the kprobe.

```
unsigned long *bp; /*breakpoint address*/
char *bad_buffer; /*buffer*/
unsigned long addr; /*temporary for incoming addr*/
struct kprobe kp; /*kprobe*/
```

Next the probe defines a function (j_write_target) that essentially overrides the default write_target function using a jprobe. This new function checks that the incoming buffer will fit inside the allocated space of the buffer and fire off a VIOLATION message if the probe is violated.

```
int j_write_target(struct file *file, const char *buffer,
unsigned long count, void *data)
{
   int len = 0;
   ...
   len = strlen(bad_buffer);
   printk("The length of the target buffer is: %d\n",
          len);
   if (count > len) {
      /* Security Violation Reaction Here */
      printk("VIOLATION!!!\n");
   }
   jprobe_return();
   return 0; /*NOTREACHED*/
}
...
/*jprobe*/
static struct jprobe my_jprobe = {
   .entry = (kprobe_opcode_t *) j_write_target
};
```

The above function is then associated with a jprobe structure that is later registered with the kernel.

Ultimately the probe needs to be setup and registered in the initialization routine (invoked by insmod). During initialization we ensure that the arguments passed in are not NULL and convert the character string representation of the addresses into the actual memory addresses.

```
static int __init init_catch_buffer_probe(void)
{
   int ret=0;
   /*Bring in the breakpoint address*/
   if (breakpoint == NULL) return -EINVAL;
   addr = simple_strtoul(breakpoint, NULL, 16);
   if (addr == 0) return -EINVAL;
   bp = (unsigned long *) addr;

   /*Bring in the buffer address*/
   ...
```

Once all of the necessary variables are initialized we setup and register the jprobe and our kprobe. While we do not need both the kprobe and the jprobe (we only need the jprobe for this example), the additional constructs are useful for future probes that we will be creating.

```
...
my_jprobe.kp.addr = (kprobe_opcode_t*) bp;
...
if ((ret = register_jprobe(&my_jprobe) < 0)) {
...
kp.addr = (kprobe_opcode_t*) bp;
...
if ((ret = register_kprobe(&kp) < 0)) {
...
```

Similar to how a Java Applet has both an init() and destroy() routine for initializing and cleaning up after an Applet, a module has both an initialization (as seen above) and an exit routine. The job of the exit routine is to clean up after the modules is removed from the system. The only work our probe needs to do is remove the two registered probes in the reverse order they were allocated.

```
unregister_kprobe(&kp);
unregister_jprobe(&my_jprobe);
```

With the above probe module loaded into the kernel we can rerun a command with a longer than expected argument and the probe should capture the exploitation of the vulnerability. Finishing with the following listing extracted from the kernel ring-buffer where all printk() commands are directed:

```
...
JPROBE_FUNCTION
The value of the incoming buffer is: \
01234567890123456789012345 6
The lenth of the incoming buffer is: 27
The length of the target buffer is: 26
VIOLATION!!!
```

5 Conclusion

In summary, our form of contract builds on the work done by Meyer[11] and Barringer *et al*[2] by adding logic for security vulnerability monitoring using contracts. Not only will CB_SAMF be able to monitor security related assertions, reactions to violations and tracking of meaningful forensic data is possible. Our model is capable of spanning multiple software architectural layers during SOAD, as our breakpoint-based contracts will be applied against multiple software entities providing a more diverse and complete approach for larger systems composed of multiple layers.

We have shown that it is feasible to implement a monitoring system based on probes which is able to capture the exploitation of vulnerabilities even at the lowest levels of the runtime architecture (the kernel). It should also be noted that in order for the above probe to be generated dynamically a tool should only have to be fed the addresses of the needed symbols in addition to the vulnerability type and code. These properties will be derived from the contract and code. Automation of many of these steps should be possible. For example, a static analysis tool could be used to provide the necessary information for the identification properties needed by the contract. Then the security tester can choose the other attributes of the contract before assertion probes are generated. Once contracts are defined the probes can be dynamically generated by extending the Systemtap scripting language and evaluation could be achieved through the employment of metrics using another probing framework. Such an approach can be used for both SOAD and monitoring SOAs during the maintenance phase of the SDLC.

Finally, in order to assess the tolerance of a given system to security violations we desire an appropriate metric. The family of time-to-intrusion metrics is promising. We propose using, in the future, a metric similar to the minimum-time-to-intrusion (MTTI) used by Voas *et al* [16]. Realization of metrics, such as MTTI, using probes has the benefit of not requiring changes to the target source code.

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Mathematics of Information Technology and Complex Systems (MITACS), and industrial partners.

References

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15(1):73–132, 1993.

[2] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Program monitoring with ltl in eagle. *ipdps*, 17:264b, 2004.

[3] S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 48–62, Washington, DC, USA, 2006. IEEE Computer Society.

[4] S. N. Chari and P. Cheng. Bluebox: A policy-driven, host-based intrusion detection system. *ACM Trans. Inf. Syst. Secur.*, 6(2):173–200, 2003.

[5] A. Février, E. Najm, and J. Stefani. Contracts for odp. In *ARTS '97: Proceedings of the 4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software*, pages 216–232, London, UK, 1997. Springer-Verlag.

[6] D. Gao, M. K. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 318–329, New York, NY, USA, 2004. ACM Press.

[7] C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in distributed systems: a specification-based approach. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, page 175, Washington, DC, USA, 1997. IEEE Computer Society.

[8] L. Lamport. A simple approach to specifying concurrent systems. *Commun. ACM*, 32(1):32–45, 1989.

[9] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[10] J. C. McKim Jr. Programming by contract. *Computer*, 29(3):109–111, 1996.

[11] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[12] D. K. Peters and D. L. Parnas. Requirements-based monitors for real-time systems. *SIGSOFT Softw. Eng. Notes*, 25(5):77–85, 2000.

[13] N. Petroni, T. Fraser, A. Walters, and W. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *15th USENIX Security Symposium*, pages 289–304, August 2006.

[14] M. Pohlack, B. Döbel, and A. Lackorzyński. Towards runtime monitoring in real-time systems. In *Eighth Real-Time Linux Workshop*, pages 173–184, 2006.

[15] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[16] J. Voas, A. Ghosh, G. McGraw, F. Charron, and K. Miller. Defining an adaptive software security metric from a dynamic software failure tolerance measure. *Computer Assurance, 1996. COMPASS '96, 'Systems Integrity. Software Safety. Process Security'. Proceedings of the Eleventh Annual Conference on*, pages 250–263, 17-21 Jun 1996.