Security Agility in Response to Intrusion Detection

Mike Petkac
NAI Labs
mpetkac@nai.com

Lee Badger
NAI Labs
lbadger@nai.com

Abstract

Cooperative frameworks for intrusion detection and response exemplify a key area of today's computer research: automating defenses against malicious attacks that increasingly are taking place at grander speeds and scales to enhance the survivability of distributed systems and maintain mission critical functionality. At the individual host-level, intrusion response often includes security policy reconfiguration to reduce the risk of further penetrations. However, runtime policy changes may cause traditional software components, designed without (dynamic) security in mind, to fail in varying degrees, including termination of critical processes. This paper presents security agility[1], a strategy to provide software components with the security awareness and adaptability to address runtime security policy changes, describes how security agility is packaged in a prototype toolkit, and illustrates how the toolkit can be integrated with intrusion detection and response frameworks to help automate flexible host-based response to intrusions.

1. Introduction

Attacks against distributed systems that compose our technological infrastructure are increasingly taking place at speeds and scales that decrease the effectiveness of human response. To increase the survivability of critical systems and maintain mission crucial functionality, much of today's computer security research, including cooperative frameworks for intrusion detection and response, seeks to develop automated defenses capable of rapid, intelligent responses to address attacks and anomalies as they occur. Two fundamental host-based responses to attacks, the primary focus of this paper, are discrete administrative actions (e.g., killing processes) and security policy reconfiguration to reduce the risk of furthers penetrations. However, dynamic policy changes may degrade a system's functionality because traditional software components, designed without (dynamic) security in mind, often exhibit shortcomings when confronted with policy changes that alter their execution environment in unexpected ways. For example, programs may crash or fail silently when critical resources are no longer accessible or they may allow the continuation of processing in violation of new security rules.

As a specific illustration, consider the TCP Wrappers [1] program commonly employed on UNIX[2] system-based hosts. This process enforces an access control policy on receipt of service requests to restrict hosts and/or users that can connect to local network services. When a runtime modification of its file-based policy restricts additional hosts/users from service use, the new policy will mediate access on subsequent service requests. However, since previous security decisions are "grandfathered", services initiated before the change may continue in violation of the new policy rules. Without human intervention to terminate violating processes, those services will persist, allowing possibly malicious entities to continue their exploits. The UNIX logging daemon, *syslogd* provides another example of a process that may be negatively affected by dynamic policy changes. *Syslogd* records important events to log files during system operation, such as failed login attempts and attempt to gain the specially privileged *root* user access. However, should an error occur while writing information to a file, as might be produced by tightening an access control policy, *syslogd* will cease attempting to output further information to that file. Thus, subsequent information, which might be critical in nature, will be lost even after proper access to the file is restored. Similar issues abound in many other processes that are sensitive to changes in security policies and functions, such as cryptographic algorithms, discretionary access control schemes and enhanced access control policies (e.g., information disclosure policies, integrity policies, and firewall configuration policies).

Security agility [2] is a software flexibility technique that extends the functionality of software components to accommodate the dynamic security properties of their environment. An agile software component (process) is aware of its security environment, is able to enforce "its

[1] This research is supported by DARPA contract F30602-97-C-0225.

[2] UNIX is a registered trademark of The Open Group.

part" of a more global policy, and contains internal mechanisms that can adapt its functionality to reliably conform to authorized policy changes. Security agility can be provided in a toolkit for software developers or, in many cases, added via wrappers [3] without recompiling software. We have developed a Security Agility Toolkit[3] for UNIX-based platforms (whose general techniques can be extended to Windows NT[4]), that combines elements of both approaches. The toolkit unobtrusively integrates into a host's environment and transparently provides a variety of prepackaged functionality for both mission-specific and standard system processes without requiring source code modification or recompilation. Additionally, the awareness and adaptive techniques provided by the toolkit can be customized to meet the specific needs of individual components and/or the security environment in which they are deployed.

The Security Agility Toolkit can also elevate a host's level of effective responses to intrusions that to date have been quite limited (e.g., kill offending processes). In part, this limited capability may reflect basic limitations in current-generation software. If components were more flexible and expressive with regard to security policies, we believe more practical response choices would be available. The toolkit provides the means to integrate such capabilities into processes to realize more flexible intrusion-tolerant systems without requiring human intervention at the critical period when attacks take place.

Our proposal to integrate intrusion detection and security agility assumes the presence of a framework for intrusion detection and response information exchange, such as proposed by the Common Intrusion Detection Framework (CIDF) working group [9] and the Intrusion Detection Working Group (IDWG) ([11], [12]) of the Internet Engineering Task Force (IETF). These frameworks include intrusion detection systems, event analysis, and response units that share information using expressive languages. The information exchanged includes general intrusion detection events (e.g., scan and probe activity, buffer overflow attacks), discrete response directives (e.g., copy, move, or delete files; suspend, resume, or terminate processes), and response directives of a more continuous nature (e.g., account auditing and message blocking). The Intrusion Detection and Isolation Protocol (IDIP) [13] provides a concrete example of an intrusion detection and response framework. In addition to tracing intruders to the point of origin, IDIP incorporates elements of the CIDF, including its Common Intrusion Specification Language (CISL) [10], to marshal response units. In such a framework, the Security Agility Toolkit can be utilized as a response unit on individual hosts to automate discrete response directives, help automate policy changes based on the information exchanged, and provide component adaptation to policy changes. We intend to construct a simple simulation testbed to demonstrate these security agility response capabilities to further enhance the survivability of systems under attack.

This paper presents an overview of security agility, discusses security agility response capabilities, and provides a scenario to demonstrate how the security agility response capabilities can be integrated with a framework for intrusion detection and response information exchange. Finally, this paper presents the implementation of the Security Agility Toolkit, discusses related work, and presents some possible future security agility enhancements.

2. Security agility and its response overview

The security policy awareness and adaptability provided by security agility was designed to be applicable to the many execution contexts and security semantics that may comprise a heterogeneous distributed execution environment. To attain this goal, the Security Agility Toolkit employs a two-fold strategy to overcome heterogeneity obstacles. First, it limits the problem space by embedding processes with pre-formulated security policy models and mechanisms to support security reconfiguration. Second, it provides a flexible component software architecture that can be extended to support additional security semantics and execution contexts and supports dynamic code extensions to add or change security-relevant behavior to maintain compatibility with new security rules. We consider some aspects of each strategy in the following paragraphs and provide additional toolkit implementation details in section 4.

Domain and Type Enforcement (DTE) [4], an elaboration of type enforcement [5], was chosen as the embedded policy model to base initial security agility research on for reasons that included its common organization with other important security models[5] and its robust UNIX prototype that supports dynamic policy change. Another important feature of the DTE kernel-based access control mechanism is its implicit attribute management framework that was developed as a non-invasive means of associating DTE security attributes on unmodified file system hierarchies. Our toolkit employs a modification of this framework to help implement an expressive and flexible embedded dynamic application-level policy model that provides components running on

[3] The prototype Security Agility Toolkit is available at ftp://ftp.tislabs.com/pub/agility.

[4] Windows NT is a registered trademark of Microsoft Corporation.

[5] DTE, the Bell and LaPadula confidentiality model [6], the Biba integrity model [7], and the Clark/Wilson model [8], share a common organization: a subject/object framework and security attributes that group subjects by their classification and objects by their type.

DTE and non-DTE systems a basis for uniform and unobtrusive security decisions. Like TCP Wrappers, the application policy model, referred to as the Access Decision Function (ADF), provides an additional layer to a "defense-in-depth" strategy. Moreover, the toolkit can reevaluate prior ADF policy decisions when its rules change to support the current environment.

The Security Agility Toolkit's architectural goals are carried out by three primary elements: an agile policy, an AGility Authority (AGA), and an agility subsystem. Together, these elements also provide much of the infrastructure to support the toolkit's intrusion response capabilities. The agile policy, which is entered on a per-host basis, specifies the embedded security models, application-level security policy rules, and dynamic code extensions for a host's agile components. The agility authority provides the policy management service for a host, including the run-time representation of its agile policy. The agility subsystem is a collection of interactive libraries that are grafted onto components to perform all the security-specific processing directed by the agile policy. The subsystem libraries contain the embedded policy models and the control framework to transition component-specific processing to the subsystem when key runtime events occur, such as access to external resources (e.g., system calls, retrieval of password information) and after policy changes take place. The subsystem's primary libraries are structured in object oriented classes to support security semantic extensibility, such as the inclusion of additional embedded security policy models or cryptographic algorithms. By coordinating agile policy modifications and underlying security policy changes (e.g., DTE) with the toolkit's agility authority, agile components' security services can be reconfigured during runtime. To assist in this coordination of security agility in distributed systems, an additional toolkit component, the AGility Authority Manager (AGAM), has been developed to provide a centralized service for policy entry and distribution[6].

Figure 1 shows our approach to integrate security agility into an intrusion detection and response information exchange framework. Response functionality will be inserted into both the agility authority and the agility authority manager components, which run as privileged processes. The AGAs will immediately invoke discrete response directives (e.g., delete file X) and coordinate continuous response directives (e.g., audit user *joe*) with agile components with minimal latency. The AGAM and the AGAs will extract response directives and other information exchanged in the intrusion detection framework (e.g., events such as probes, scans, and specific attacks) into additional security policy rules

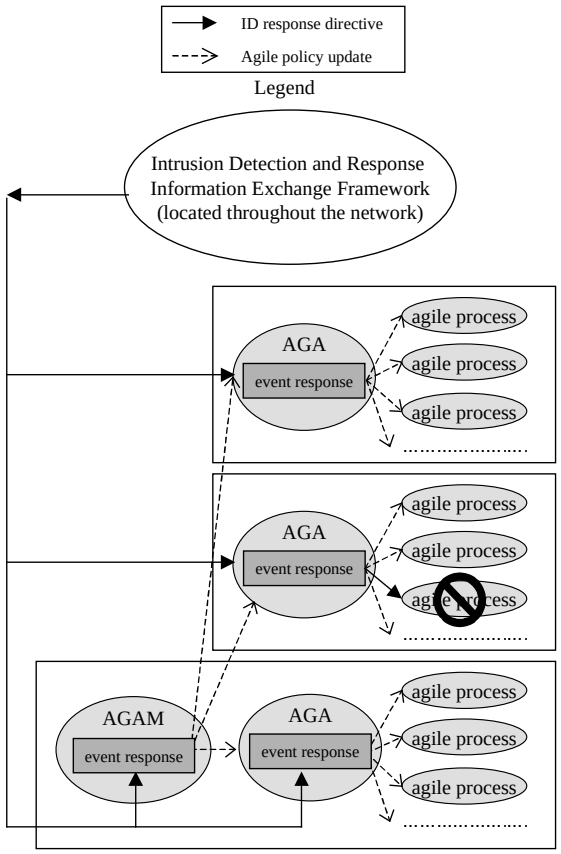

Figure 1: The security agility toolkit as intrusion detection response units

and dynamic code extensions. Policy changes directed by the AGAM will be forwarded to the appropriate hosts' agility authorities in the distributed system, while the AGA will monitor events for policy changes that will take effect on the host it serves. The dynamic policy changes produced by this approach can help safeguard the entire system against related attacks and extend detection and deterrent capabilities.

While it is clear that a privileged process can be designed to implement many discrete response directives, the more interesting and challenging portion of integrating the Security Agility Toolkit into intrusion detection and response frameworks lies in the policy reconfiguration capabilities the toolkit can provide. There are a number of questions concerning policy reconfiguration that must be answered, including:

- How does security agility make components more flexible and expressive with regard to security policies to enable more practical response choices?
- How can the toolkit use general response directives and additional detection and response information to enable defensive security policy changes?

[6] The prototype's runtime policy distribution is not secured. For an operational system, authentication and encryption would be added.

Rule Type	Description of Rule Statement						
ADF Policy	**action** deny deny except	**what/who on behalf of who/what** processes, users, hosts	**access** read, write, execute, system call, root privilege, etc.		**(from/for/to/using) what/who** object, processes, users, hosts		**when** always, activation criteria
Dynamic Code	**action** run (extension, callback function)	**on** access, access failure, policy change	**of** system call	**to** object, object label	**in place of** function name	**for** process, user	**when** always, activation criteria

Figure 2: Agile policy rule sets

- How can the Security Agility Toolkit assist in supporting the continuation of vital mission processing when defensive policy changes occur?
- How can automated policy changes be implemented with confidence?
- Can the toolkit fully automate policy changes based on information exchanged in an intrusion detection and response framework?

The next section addresses these questions by describing the toolkit's policy reconfiguration response capability and providing a scenario to illustrate their use.

3. Security agility intrusion response

Our goal in proposing to integrate the Security Agility Toolkit into an intrusion detection and response framework is to provide automated policy changes in response to intrusions for many execution environments. In the context of the toolkit, policy changes refer to both embedded security policy modifications and reconfiguration of dynamic code extensions to adapt to new policy states. It is this combination of policy awareness, enforcement, and adaptation the toolkit provides that enables more practical response choices. Since the Access Decision Function (ADF) is applicable to heterogeneous environments, our discussion of dynamic security policy changes will focus on the ADF policy model, although similar techniques could be developed for other dynamic security policy models, such as DTE, that are specific to more limited environments. We provide a closer look at the agile policy introduced in section 2 since it provides the complete specification of policy changes we will consider.

A host's agile policy specification consists of the construction of policy "statements" from pre-formulated rule-sets to direct the agility subsystem's security relevant component processing. As shown in Figure 2, the rule-sets are classified into two primary categories: the ADF policy and dynamic code extension specifications. Figure 2 shows the general text form of the rule-sets used to construct policy statements, although the statements are actually written using an agile policy language representation of the text. The rule-sets consist of an action item and a description of the action, though not all the description fields will be relevant for all actions. Each rule-set will be extended to include a *when* description field, or activation criteria, to help automate the enforcement of rules based on identified conditions (e.g., "deny except tom root privilege when ID event X is received"). The following paragraphs examine each category of agile policy rule-sets.

ADF policy - The ADF security policy provides agile components with application-level access control based on the actions *deny* and *deny except*. A "deny except" action is the equivalent of an "allow only" option to express limited access rules that are difficult to specify with the deny action (e.g., allowing only the administrator write access to password files). Security attribute association is an important part of security specification and enforcement for any policy model, including the ADF model, though it is often an administrative "killer" in security enhanced systems. However, the ADF policy representation leverages a modification of the Domain and Type Enforcement (DTE) [4] implicit attribute management framework to unobtrusively associate security attributes with hierarchically organized objects (e.g., directories and files) in the address space of an agile process. To help support fine-grain ADF rule statements for intrusion response, the ADF policy extends the DTE implicit labeling of objects to allow labels to be associated with specific file types (e.g., binary files, shell scripts, etc.) and supports time-activation and deactivation of

labels. Combining the description and object labeling, the ADF policy can be used, for example, to deny the execution of binaries for an hour after creation.

dynamic code extensions - Dynamic code extensions provide components with security policy awareness, enforcement, and adaptive functionality and can also extend a component's processing for such events as auditing or encryption. *Run* is the only action directive for this rule-set, however, it is not required to implement every dynamic code extension. For instance, when ADF policy statements are associated with a process, its agility subsystem will automatically invoke internal code extensions that consult the embedded policy model for access decisions. ADF policy enforcement is an example of the toolkit's prepackaged code extensions that provide default functionality without requiring any further development. Additional default functionality includes embedded auditing capabilities, resource-state tracking to revisit grandfathered decisions on policy change, and default adaptive behaviors to implement when resource accesses are denied (e.g., wait until available, ignore errors, internally buffer data, terminate, suspend, and resume processing). Dynamic code policy statements can take advantage of the toolkit's embedded default functionality, supplement default processing by associating custom functions with resources, or use the "*in place of*" statement field to completely replace the default functionality associated with specific function calls with customized processing. Using the UNIX system logging function *syslogd*, a statement that uses default processing might look like "run *wait* on access failure of all system calls to all resources for **/usr/sbin/syslogd**". Similarly "run **/agility/msgs:msg_callback** on access failure of write to **/var/log/messages** for **/usr/bin/syslogd**" would invoke the *msg_callback* function when a write to *messages* file fails. The *msg_callback* function might be used to write log events to a secondary file until the original **/var/log/messages** file is again available.

Insightful code extensions will be a key in enabling defensive security policy changes that may be required to respond to general intrusions and anomalies. For example, extensions to suspend, resume, or terminate connection-oriented processes may be included as statements in a defensive policy specification that automatically trigger according to specified activation criteria. However, UNIX servers that validate connection attempts (e.g., *telnetd* or *rshd*) typically authenticate a client's credentials without retaining the information. Dynamic code extensions to capture and retain this information for such processes will be needed to re-authenticate connections when security policy changes occur. Similarly, allowing critical processes to continue functioning to the fullest extent possible when policy changes occur can be accomplished by introducing well defined dynamic code extensions, such as those that use alternative resources and/or algorithms (e.g., the previous paragraph's /**agility/msgs:msg_callback** function).

The toolkit will automate agile policy changes to counter intrusion detection events to the greatest extent possible. Toolkit functionality can be directly used to automate a subset of policy changes, (e.g., suspend all processes for user *sue* when a response directive specifies termination of a user *sue* process, enable fine-grain auditing when a directive orders an account audit of user *joe*). For such constrained policies, automation consists of specifying a set of rules that adequately cover a policy space of concern, and automatically triggering the rules when their enabling conditions become true. Complete automation of less constrained changes, however, is a subject of ongoing research. For example, complete automation of policies that inject new code into running components would require significant advances in software specification and composition techniques. To increase adaptability without waiting for such changes, we have formulated an approach to transition an agile system between pre-specified comprehensive agile policies.

Our approach is to provide a three-tiered strategy to allow an operator to specify the agile policy that will be enforced based on the current environment. First, it will allow the operator to define the agile policy in multiple levels (e.g., normal or alert) that contain rule-statements to address varying degrees of security postures. Second, the activation criteria of the agile policy rule-statements will support specification of explicit rules to be automatically enforced when anticipated events occur. This will reduce the burden on operator interaction to activate policy change when attacks are taking place at machine speed. Third, it will support rule-statement templates that the agility authorities and the agility authority manager will complete based on intrusion responses received.

The three-tiered agile policy specification approach provides the basis for defining defensive security policies that can respond to general intrusions and anomalies that are not specified in response directives. It allows the operator to fashion policy levels based on known and followed practices used to attack systems. By monitoring intrusion detection exchange traffic for events and evaluating their ramifications, the operator can manually initiate a change of policy level. Optionally, activation

criteria could be associated with the operator-defined policy levels.

The next section presents a scenario to illustrate our security agile policy specification approach and the intrusion response capabilities of security agility.

3.1. Security agility intrusion response scenario

Buffer overflow attacks against well-known services continue to be a primary vulnerability for system penetration. A visit to the CERT Coordination Center[7] web site illustrates the need for concern. It states: "We receive many daily reports of scanning and probing activity. The most frequent reports tend to involve services that have well-known vulnerabilities. Hosts continue to be affected by exploitation of well-known vulnerabilities in these services."

To illustrate how the three-tiered agile policy specification approach might be implemented, we formulate a simple defensive security policy that monitors and reacts to attack and probe events that are normally associated with attempts to penetrate systems using buffer overflow attacks. This policy must carefully balance its defensive strategies so that systems vulnerabilities are sufficiently reduced to safeguard against anticipated attacks and counter actual attacks, but maintain critical services and functionality to the fullest extent possible.

We will consider a scenario where an operator defines three levels of security: *normal*, *alert*, and *survive*. The high-level strategy to implement these three-levels of security is for the *normal* policy to allow a relatively permissive environment when detection information indicates little hostile intent. The *alert* policy will supplement any *normal* policy restrictions with additional policy statements and activation criteria to increase component awareness and response capabilities when probe and scan information indicates a higher probability of a pending attack. Finally, the *survive* policy will build on the alert policy statements to quickly enforce stringent countermeasures to suppress penetrations on the attacked hosts, and safeguard other hosts from similar attacks. Additionally, specific response directives will supplement the agile policy in effect. Although tightening of security rules could hamper the mission objectives of the system, security agility's adaptive behavior to dynamic policy changes can minimize lost functionality during restrictive policy periods. Careful specification of activation criteria and thresholds associated with activation criteria can also ensure that the policy level and its active statements are sufficient to address the security needs of the current situation without being overly restrictive. To complete the policy discussion, we concentrate on detailing some specific policy statements that might comprise each of the *normal*, *alert*, and *survive* policy levels and the activation criteria that may be applied to automate statement enforcement and policy level transitions.

Figure 3 presents our policy specification[8]. The statements contained in the policy are associated with the UNIX internet "super-server" *inetd* initiated services which are often the target of buffer overflow attacks. The *normal* policy can be thought of as the base policy. Its most important statement, in the context of this discussion, is a dynamic code extension to retain connection information for *inetd* spawned services. While the connection information serves no purpose at this policy level, it may be relevant to a statement in the *alert* or *survive* levels should a transition to either policy level occur to enable suspension or termination of processing.

Activation and deactivation criteria[9] are specified in the policy to transition from the *normal* to the *alert* policy and visa versa. The policy level activation criteria, which the AGility Authority Manager (AGAM) will monitor, looks for scans and/or probes that could raise the likelihood of a remote break-in attempt. The criteria assumes the presence of data, such as specified in the Common Intrusion Specification Language (CISL) [10], to identify the class of information, the certainty of the event, and the host target the event applies to. Criteria to activate to and deactivate from the *alert* policy should be set so a continuous transitioning between policy levels does not occur. In general, if a situation warrants policy constriction, the more restrictive policy should continue to be enforced until it is clear the threat has subsided.

The statements contained in the *alert* policy, shown in Figure 3, provide a subset of the additional restrictions that would likely be introduced in a more complete policy. However, they further demonstrate the defensive policy reactionary approach, including additional dynamic code extensions to provide supplement auditing and statement activation criteria to automatically suspend suspicious connections as well as ADF statements to deny future connections from questionable sites. The statement activation criteria associated with two of the policy statements, one which might well make use of connection information collected by the *normal* policy's dynamic code extension, are slight modifications of the activation criteria to activate to and deactivate from the *alert* policy. Activation criteria for statements will be monitored by a host's AGA, which may have to coordinate activation criteria, as illustrated by the dynamic code extension's suspend action, with agile components. The more

[7] "CERT" and "CERT Coordination Center" are registered in the U.S. Patent and Trademark Office.

[8] For readability, the policy is represented in text form rather that its agile policy language representation.

[9] The activation criteria used in this model are consistent with Intrusion Detection Exchange Format Internet draft proposed by IETF's IDWG.

key: **ADF** - Access Decision Function (application-level policy)
DCX - Dynamic Code eXtension

***normal* policy** - base ADF policy statements plus DCX that retains connection information for inetd spawned services

activation criteria to *alert* policy: X_1 number of accepted intrusion detection events are received in Y_1 seconds

- accept event if:
  1) its class is contained in a specified list of probes/scans (e.g., finger probe, port scan, mount scan, etc.) and
  2) its *certainty* exceeds a specified minimum threshold value

deactivation criteria to *normal* policy: X_2 number of accepted intrusion detection events are not received in Y_2 seconds (event acceptance conditions match activation criteria to alert policy)

***alert* policy** - *normal* policy statements plus:

- DCX to:
  - enable suspension of inetd spawned services,
  - audit all inetd spawned connection attempts and child sessions
  - suspend active inetd spawned connections when an accepted event is received
    - accept event if:
      1) its class is contained in a specified list of probes/scans and
      2) its *certainty* exceeds a specified minimum threshold value and
      3) its source host address matches the connection request source address
- ADF statement to:
  - deny host(s) connection access to inetd spawned services when an accepted events is received
    - accept event if:
      1) its class is contained in a specified list of probes/scans and
      2) its *certainty* exceeds a specified minimum threshold value
- additional policy statements to audit/restrict services

activation criteria to *survive* policy: alert of any buffer overflow attack

deactivation criteria to *alert* policy: Not applicable, requires manual deactivation

***survive* policy** - alert policy statements plus:

- DCX to terminate active inetd spawned sessions of attack host
- ADF to:
  - deny connection access for attack host
  - deny except administrator access to compromised service

Figure 3: Agile policy example

restrictive *alert* policy simply adds to the rule statements contained in the *normal* policy. This ensures that a properly formed base policy remains intact.

Transition to the most restrictive policy level, *survive*, can once again automatically occur whenever a buffer overflow attack event is received. However, in this example, transition from the survive policy to a lower policy level will require an operator to be initiated.

The security agile policy specification approach presented in this section can be applied to other policy specification techniques. For example, Domain and Type Enforcement (DTE) supports policy specification and extensions to be encapsulated in *modules*. DTE defines strict rules that apply to proper module formation to maintain the intent of the original policy. With some extension, activation criteria could be associated with DTE modules or even DTE policy rules within modules.

4. The security agility toolkit implementation

In this section, we highlight the prototype Security Agility Toolkit to explain how its previously discussed techniques are implemented. The toolkit was initially developed on the now outdated UNIX system-based BSD/OS[10] 2.1 operating system that hosted Domain and Type Enforcement (DTE) (and is described in a previous paper [2]), but is now implemented on up-to-date FreeBSD and Linux environments. While the primary architecture of the toolkit has remained consistent across platforms, the Executable and Linking Format (ELF) binary format employed by FreeBSD and Linux supports a more transparent implementation of agile techniques than that offered by the *a.out* binary format of BSD/OS. In ELF environments, security agility is implemented without any system or component source code modification or recompilation. This allows the toolkit to be unobtrusively integrated into previous, current, and future FreeBSD and Linux releases and facilitates porting agility to other UNIX variants, such as Solaris, even if system or component source code is unavailable.

The design of the Security Agility Toolkit closely follows the two-fold strategy presented in section 2. It limits the problem space using the DTE[11] and Access Decision Function (ADF) embedded security models but provides a flexible and extendable architecture that supports dynamic code extensions to add or change

[10] BSD/OS is a trademark of Berkley Software Design, Inc.

[11] A (nearly complete) version of DTE now exists for the FreeBSD 3.2 operating system and is being ported to FreeBSD 5.0.

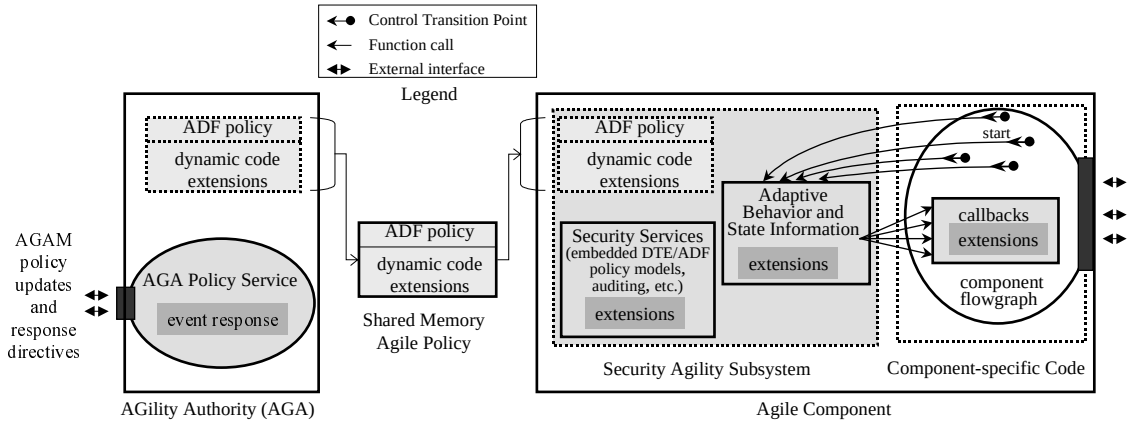

Figure 4: Security agile component architecture

security-relevant behavior to maintain compatibility with new security rules. Figure 4 depicts the general architecture of a security *agile* component and displays the interaction of the key elements of the security agility toolkit: the AGility Authority (AGA), the agile policy, and the (security) agility subsystem. The component-specific code represented by the agile component's oval in Figure 4 implements the component's non-security responsibilities. The security-specific functionality is carried out by the agility subsystem.

The agility subsystem contains the toolkit's embedded policy models and its framework to employ dynamic code extension. The subsystem is primarily composed of object-oriented C++ libraries to provide the base classes for additional embedded policy models and security semantic extensions, to take advantage of standard compiler and linking techniques for unobtrusive integration with components, and to support cross-platform development and porting. The "extensions" shown in Figure 4 depict the dynamic code extensions the agile policy directs a component's security agility subsystem to employ. The agility subsystem utilizes two mechanisms to invoke dynamic code extensions that are also displayed in Figure 4: Control Transition Points (CTPs) and security-relevant callback functions.

Control transition points intercept component function calls to transfer control to the agility subsystem during normal component operation. CTPs are implemented in dynamically loadable libraries that overlay the libraries containing the original function calls. The CTP libraries are automatically generated during toolkit compilation from files that prototype the functions to be intercepted and are compiled dependent on their original library as well as the other agility subsystem libraries. Dynamically linked binaries load the CTP libraries and their dependencies during process startup to inject the agility subsystem into components. By default, the toolkit associates control transition points with the C library routines that bracket access to external resources, in effect making most dynamically linked processes agile by default. However, the toolkit supports customization of the C library CTPs through modification of the prototype file as well as creation of CTP libraries for any system or custom library. The default functionality and behaviors provided by the toolkit, as discussed in section 3, are embedded in the CTP library that overlays the system's C library. Security-relevant callback functions are invoked by the agility subsystem on notification of a security policy update. Although invoked at different periods of component processing, the mission of CTPs and callback functions often overlap: both may provide security awareness and adaptive behavior for an agile component.

At system startup, a host's AGA is initialized to parse the host's agile policy file into a shared memory representation for agile processes and create sockets for external communications (e.g., for policy updates and response directives). The AGA updates the shared memory policy structures as policy changes are received but will leave the previous policy intact to assist the components' agility subsystems in addressing policy change synchronization (e.g., to mediate a server's decision to a client's request whose policy has not yet been updated). The agility authority will also monitor the (de) activation criteria of the active policy as intrusion detection events are received to (de) activate ADF and dynamic code specification statements.

When an agile process begins execution, control passes to the agility subsystem using common ELF and C++ language/compiler techniques before executing any of the original component code. The subsystem's initialization routine configures the component according to its agile policy by loading all relevant dynamic code extensions and determining the Access Decision Function rules that apply to it. During component operation, each occurrence of a library call that is wrapped by a control transition point will transfer processing control to the

agility subsystem. When the agility subsystem receives control, it checks if a policy update occurred since the subsystem last returned processing to component-specific code, and, if so, reconfigures the component accordingly. Dependent on the type of policy change received (e.g., security policy model or dynamic code extensions) this reconfiguration might include reevaluation of previous security decisions, invocation of custom or default security specific callback processing, and reconfiguration of dynamic code extensions. After completing its policy change processing, the agility subsystem will invoke any default or custom processing specified for the control transition point. If no additional processing is specified, the original library call simply will be invoked and its results will be returned to the calling function.

Although our discussion has focused on UNIX system-based hosts, a Windows NT version of the security agility toolkit has been explored, though not fully developed. With some success, the NT toolkit attempted to inject the agility subsystem onto processes using a modified registry entry and a technique to patch the import address table described in [14] to add the agility subsystem processing. Finally, the current UNIX-based toolkit implementation is not applicable to ELF statically linked binaries. However, the BSD/OS toolkit employed *libdld*, a GNU General Public License library, to incorporate agility techniques with static binaries. Development of a similar library may be able to support integration of security agility with statically linked ELF binaries.

5. Related work

Our work relates most closely to work in intrusion response, and adaptive systems.

The Common Intrusion Detection Framework (CIDF) working group ([9], [10]) and the Internet Engineering Task Force (IETF) working group ([11], [12]), named the Intrusion Detection Working Group (IDWG), provide a framework for intrusion detection and response information exchange. The Intruder Detection and Isolation Protocol (IDIP) [13] focuses on tracing intruders to their points of origin and marshalling response components. A number of Generic Software Wrappers [4] have been written to respond to IDIP messages by suspending processes, revoking users, etc. Firewalls can also be used as response components, by restricting services or remote clients. A variety of response techniques are possible. For example, [15] employs a system-call interception layer and proposes intruder isolation using active networks.

A number of efforts focus on extensible, or adaptable, systems. The SPIN system [16] allows code to be dynamically loaded into the operating system kernel, and controlled via type-safety, while [17] applies domain/type techniques to further control dynamic kernel extensions. The Cactus system [18] builds on the x-kernel to provide a high level of service and service-implementation configurability. The MARX system [19] applies market-based techniques to help software components survive resource shocks. Work at the Oregon Graduate Institute [20] has investigated the use of adaptation spaces, which are spaces of alternative implementations for functions within a component along with conditions that might prompt a change. This is a generic framework that seeks to express well-formed adaptations. Several techniques also add security policy enforcement after-the-fact. The Naccio [21] system allows a safety policy to be added at a wrapper library layer in applications. The SASI [22] work edits byte-codes at load-time to add Java 2 stack-inspection policies to programs. Recent developments in the Flask operating system architecture [23] enable support for a wide range of security policies, including the revocation of previously granted access rights.

These techniques differ primarily from security agility in that the toolkit combines a management infrastructure (the agility authority) with a predefined notion of security policy rules and dynamic code extensions for adaptive behavior specification. The toolkit also offers extensibility for additional security semantics and application to system and custom mission components.

6. Future directions

Although the Security Agility project concluded in September 2000, there are many additional features we hope to independently add to the toolkit to increase its robustness and simplify its management. For instance, incorporation of and experimentation with additional security semantics and optimization of the agility subsystem's performance would certainly enhance the toolkit, as would an agility authority manager GUI to assist in initial policy specification and visual analysis of the runtime situation to help guide dynamic policy modifications that deviate from the current policy.

7. Conclusions

Attacks against distributed systems are increasingly taking place at speeds and scales that lessen the effectiveness of human response. To combat such attacks, much of today's computer security research seeks to develop automated defenses capable of rapid, intelligent responses. This paper has presented a strategy to help automate host-based responses to intrusions by combining security agility with cooperative frameworks for intrusion detection and response, such as that demonstrated by the Intrusion Detection and Isolation

Protocol (IDIP) [13], to realize more flexible, intrusion-tolerant systems. In addition to discrete host-based responses (e.g., kill process X), we explained how security agility can automate a second host-level intrusion response: security policy reconfiguration on constrained policies. This automation consists of specifying a set of rules to adequately cover a policy space of concern, and then automatically triggering the rules when their enabling conditions become true. Our approach provides the basis for defining multi-level defensive security policies to respond to general intrusion detection events. Although this paper presented automated policy changes in the context of a single security policy model, the prototype Security Agility Toolkit is intended to be a general tool that can be extended to conform to additional security models and semantics.

References

[1] W. Venema, "TCP Wrapper: Network Monitoring, Access Control, and Booby Traps," in *Proceedings of the 3rd UNIX Security Symposium*, Baltimore, MD, September 1992.

[2] M. Petkac, L. Badger, and W. Morrison. "Security Agility for Dynamic Execution Environments," In *Proceedings of the DARPA Information Survivability Conference and Exposition*, Hilton Head, SC, January 2000.

[3] T. Fraser, L. Badger, and M. Feldman. "Hardening COTS Software with Generic Software Wrappers." In Proceedings of the 1999 IEEE Symposium of Security and Privacy, Oakland, CA, May 1999, p. 2.

[4] L. Badger, D. F. Sterne, D. L. Sherman, and K. M. Walker. "A Domain and Type Enforcement UNIX Prototype," In *USENIX Computing Systems* Vol 9, No. 1, Winter 1996.

[5] W. E. Boebert and R. Y. Kain. "A Practical Alternative to Hierarchical Integrity Policies," In *Proceedings of the 8th National Computer Security Conference*, 1985.

[6] D. E. Bell and L. LaPadula. *Secure Computing System: Unified Exposition and Multics Interpretation.* Technical Report EDS-TR-75-306, Electronics Systems Division, AFSC, Hanscom AF Base, Bedford, MA, 1976.

[7] K. J. Biba. *Integrity Considerations for Secure Computing Systems*. Technical Report ESD-TR-76-372, AFSC Electronics Systems Division, Bedford, MA, 1977.

[8] D. D. Clark and D. R. Wilson. "A Comparison of Commercial and Military Computer Security Policies," In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, Oakland, CA, 1987.

[9] P. Porras, D. Schnackenberg, S. Staniford-Chen, M. Stillman, and F. Wu. "The Common Intrusion Detection Framework," CIDF working group document, http://www.gidos.org.

[10] R. Feiertag, C. Kahn, P. Porras, D. Schnackenberg, S. Staniford-Chen, and B. Tung. "A Common Intrusion Specification Language", CIDF working group document, http://www.gidos.org.

[11] M. Wood, "Intrusion detection exchange format requirements," Internet Draft, Internet Engineering Task Force, July 1999.

[12] H. Debar, M.-Y. Huang, and D. Donahoo. "Intrusion Detection Exchange Format Data Model," Internet Draft, Internet Engineering Task Force, January 2000.

[13] D. Schnackenberg, K. Djahandari, and D. Sterne. "Infrastructure for Intrusion Detection and Response," In *Proceedings of the DARPA Information Survivability Conference and Exposition*, Hilton Head, SC, January 2000.

[14] M. Pietrek. Windows 95 System Programming Secrets. IDG Books Worldwide, Inc. 1995.

[15] T. Bowan, D. Chee, M. Segal, R. Sekar, T. Shanbhag, and P. Uppuluri. "Building Survivable Systems: An Integrated Approach based on Intrusion Detection and Damage Containment," In *Proceedings of the DARPA Information Survivability Conference and Exposition*, Hilton Head, SC, January 2000.

[16] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. "Extensibility, Safety and Performance in the SPIN Operating System," In *Proceedings of the 15th ACM Symposium on Operating System Principles* (SOSP-15), Copper Mountain, CO, December 1995, p 267.

[17] D. Hollingsworth, T. Redmond, and R. Rice. "Security Policy Realization in an Extensible Operating System," In *Proceedings of the DARPA Information Survivability Conference and Exposition*, Hilton Head, SC, January 2000.

[18] M. Hiltunen, R. Schlichting, C. Ugarte, and G. Wong. "Survivability through Customization and Adaptability: The Cactus Approach," In *Proceedings of the DARPA Information Survivability Conference and Exposition*, Hilton Head, SC, January 2000.

[19] J. Eggleston, S. Jamin, T. Kelly, J. MacKie-Mason, W. Walsh, and M. Wellman. "Survivability through Market-Based Adaptivity: The MARX Project," In *Proceedings of the DARPA Information Survivability Conference and Exposition*, Hilton Head, SC, January 2000.

[20] S. Bowers, L. Delcambre, D. Maier, C. Cowan, P Wangle, D. McNamee, A.-F. Le Meur, and H. Hinton. "Applying Adaptation Spaces to Support Quality of Services and Survivability," In *Proceedings of the DARPA Information Survivability Conference and Exposition*, Hilton Head, SC, January 2000.

[21] D. Evens and A. Twyman. "Flexible Policy-Directed Code Safety," In *Proceedings of the 1999 IEEE Symposium of Security and Privacy*, Oakland, CA, May 1999.

[22] U. Erlingsson and F. Schneider. "SASI Enforcement of Security Policies: A Retrospective," In *Proceedings New Security Paradigms Workshop 1999.*

[23] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Anderson, and J. Lepreau. "The Flask Security Architecture: System Support for Diverse Security Policies," In *Proceedings of the Eighth USENIX Security Symposium*, Washington, DC, August 1999, p. 123.