Towards Effective Security Policy Management for Heterogeneous Network Environments

Lawrence Teo and Gail-Joon Ahn[†]
University of North Carolina at Charlotte
{lcteo.gahn}@uncc.edu

Abstract

In this paper, we overview a system-driven policy framework called Chameleos-x and discuss how a practical, system-driven approach could be used to address the problem of enforcing security policies consistently in a changing, diversity-rich environment. The Chameleos-x framework is specially designed to facilitate the management of consistent security policies in heterogeneous environments. We also describe our experimentation of Chameleos-x to demonstrate the feasibility of the proposed approach.

1. Introduction

It is extremely important to define and enforce an effective security policy for businesses and organizations that heavily rely on computer networks and information systems for their daily operations. Due to this ever-increasing reliance on computer systems, it is also critical for organizations to implement a carefully-designed security policy for their computerized environments. This brings up an important question: how do we ensure that the security policy actually works, and that it is effective at stopping attacks? One way is to evaluate the systems for conformance to the security policy. However, evaluation has its own challenges as well. As the size of an organization grows, so do its computer networks and information systems. This growth tends to introduce diversity and heterogeneity into the network, especially as new operating systems, network devices, and security technologies are adopted.

In this paper, we discuss the overview of a system-driven policy framework called Chameleos-x [6] and describe how

a practical, system-driven approach could be used to address the problem of enforcing security policies consistently in a changing, diversity-rich environment. The Chameleos-x framework is specially designed to facilitate the management of consistent security policies in heterogeneous environments. Our prior works clearly demonstrated how we could develop a policy language to specify access control policies across different operating systems [5]. There are few related works. Ponder [2] is a policy specification language for distributed systems. It is also a flexible language that targets a number of different systems. The difference between Ponder and our work is that Ponder is strictly a policy specification language, while Chameleos-x is involved in both policy specification and enforcement. Also, Chameleos-x has pluggable policy sets to support multiple system entities in large and heterogeneous environments and also provides a facility to help evaluate specified policies. Woo and Lam [7] designed a flexible language that used default logic to model authorization rules. The Authorization Specification Language (ASL) [3] is a flexible and expressive language that can be used for multiple access control policies. Related projects also include logical access control frameworks [1].

In Chameleos-x, we attempt to deploy comprehensive security policies for various types of systems. Also, we use a three-pronged strategy to enforce policies for those systems – (1) Chameleos-x assists in the *configuration* phase of the policy deployment process; (2) Chameleos-x allows the *evaluation* of policies so that the organization can be confident that the policies work as expected; and (3) Chameleos-x has a *response* mode, which refers to the ability of the Chameleos-x architecture to proactively enforce the policy based on changing conditions and events. Chameleos-x benefits organizations by introducing numerous cost and time savings. Users would be able to use a language with a common syntax to design the security policy for heterogeneous systems. They would not have to relearn a different syntax in order to deploy the same policy from one system to the next. System designers and network engineers benefit by being able to design more secure and reliable systems

COMPUTER
SOCIETY

and networks as the result of a systematic policy deployment process.

This paper is organized as follows. We discuss the design, architecture, and realization of Chameleos-x in Section 2, followed by a discussion of our experiments and results in Section 3. Section 4 concludes the paper.

2. Policy Framework: Chameleos-x

The Chameleos-x policy framework can be implemented using two major components: a policy specification language and a policy enforcement architecture. In this section, we describe the terminology used to describe the components in Chameleos-x, and discuss the Chameleos-x language and architecture.

2.1. Language and Architecture

In Chameleos-x, we introduce several new terminologies. It includes operation modes, session, context, and context library. Chameleos-x is designed to work with three high-level *operation modes*: Configuration, Evaluation, and Response. A *session* represents a typical period of time when Chameleos-x is run. A *context* can refer to an entity on just one single system, or span across multiple systems. To facilitate the definition of contexts, Chameleos-x supports *context libraries*. A context library, as its name implies, is a collection of contexts related to a particular domain. The Chameleos-x policy framework includes a language component that is used for policy specification. It is intended to cover many notions that are used to specify policies, including basic access control concepts. We have developed the basic grammar of the Chameleos-x language in Extended BNF (EBNF). Due to space limitations, we omit the EBNF grammar specification in this paper. The three major components of the Chameleos-x architecture are the Management Console, Translator, and Enforcement Monitor. The Management Console is a central management interface operated by the evaluator. It is used to "push" Chameleos-x policies to various hosts that are running the Chameleos-x Enforcement Monitor. The Management Console also specifies which operation mode should be used in each session. The Chameleos-x Enforcement Monitor is a daemon that runs continually in the background on systems that are part of the Chameleos-x framework (that is, the servers, firewalls, and IDSs). Its responsibility is to receive the Chameleos-x policy from the Management Console, and apply it on its host system. To do so, the Chameleos-x policy would have to be translated. This translation process is done by the Chameleos-x Translator, which is used to convert the Chameleos-x policy into one or more system-specific policies. The Translator resides together with the Chameleos-x Enforcement Monitor
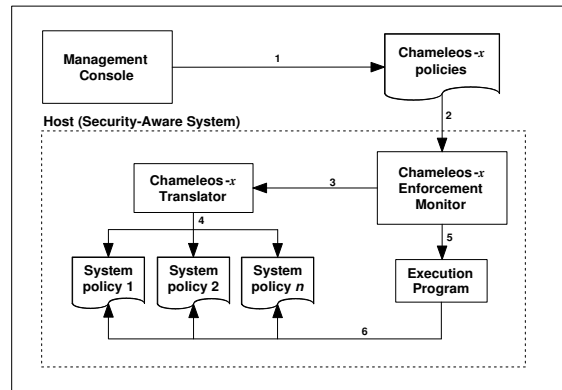

Figure 1. The Chameleos-x architecture.

(it could either be part of the Monitor, or a separate entity that is invoked by the Monitor). Each Chameleos-x variant would have its own Translator. For instance, if we are working with the Snort IDS, the Chameleos-`ids` Translator will convert the Chameleos-`ids` policy into a Snort configuration file. The layout of the components in the Chameleos-x architecture is shown in Figure 1.

2.2. Realization of Chameleos-x

In this section, we discuss our ongoing work that we are carrying out towards a full implementation of Chameleos-x. The first implementation decision we have to make is the language that we should use to implement the Chameleos-x components like the Chameleos-x Enforcement Monitor and the Chameleos-x Translator. Due to the multi-platform nature of Chameleos-x, Java would sound like the natural choice as the development language for the components, especially the Monitor. However, we chose to implement the components in portable C instead, since the Monitor would need to access low-level services on the hosts, which is something that Java does not offer conveniently. The Translator was implemented using Perl, while the extension routine libraries were developed using Perl and Bourne shell scripts, where appropriate. These components ran on the three UNIX systems: Linux, FreeBSD, and OpenBSD. In addition, our management console used NetBSD as its operating system. The decision to use different platforms was deliberately made in order to test Chameleos-x's ability to function in heterogeneous environments.

As mentioned in the language and architecture of Chameleos-x, the Chameleos-x policy framework uses contexts to describe various entities that are affected by the configuration and evaluation processes. Contexts are stored in context libraries to facilitate the creation of new Chameleos-x policies. While we could develop our own context library, for the purposes of experimentation in this section, we

chose to select a context library that is ubiquitous on UNIX systems – the `/etc/services` file. Each Chameleos-x policy can be divided into three sections, which relates to the three operation modes: Configuration (`config`), Evaluation (`eval`), and Response (`response`). The correct section of the policy is invoked depending on which of these three operation modes is being used in a typical session. For brevity, we mainly discuss how the Chameleos-`firewall` policy can be constructed and realized in real systems.

The Chameleos-`firewall` policy in Figure 2 (a) has three sections: Configuration, Evaluation, and Response. Firewalls tend to follow either a default-deny or default-allow policy, depending on the nature of the organization. For instance, an organization that favors a more restricted and closed environment, such as the military, tend to opt for a default-deny policy. More open environments like academic institutions may opt for a default-allow policy. To support these policies, we have implemented a `global_policy()` function which accepts one of two constants: `CHL_FW_DEFAULT_DENY` (for default-deny) or `CHL_FW_DEFAULT_ALLOW` (for default-allow). We also use a `define_group()` function which allows groups to be defined. In this example, we have defined a group called *blacklist* that consists of the IP address 10.0.0.3. Note that we can actually include more than one IP address in a group; we are just using a single IP address in this case to simplify our discussion. In the Configuration section, we deny access to the *ftp* context from the *blacklist* group by using the `deny_context()` function. We allow access to the *http* context from everyone else by using the `allow_context()` function with the constant `CHL_FW_ALL`.

In terms of implementation, we used a directory tree to store the extension routines shown in Figure 2 (b). In the diagram, Chameleos-`firewall` has two extensions: *pf* and *iptables*. Each routine in the extension (init, global_policy, etc.) was implemented as either a Perl script or Bourne shell script. The idea is to have these scripts generate a configuration file for their respective extensions. For instance, in the case of *pf*, the Translator would translate the Chameleos-`firewall` policy in Figure 2 (a) by converting it into a configuration file that *pf* can load (using the pf.conf syntax). On the other hand, if *iptables* was used, the translator would generate a shell script with the relevant `iptables` commands according to the original Chameleos-`firewall` policy.

The actual translation process is outlined in Figure 2 (c). The initialization process is in charge of initializing system-specific variables, as these may differ from system to system (for example, Linux usually uses "eth0" as the generic name for the first network interface, while the BSD systems tend to use driver-based names, such as "xl0" or "dc0"). The next s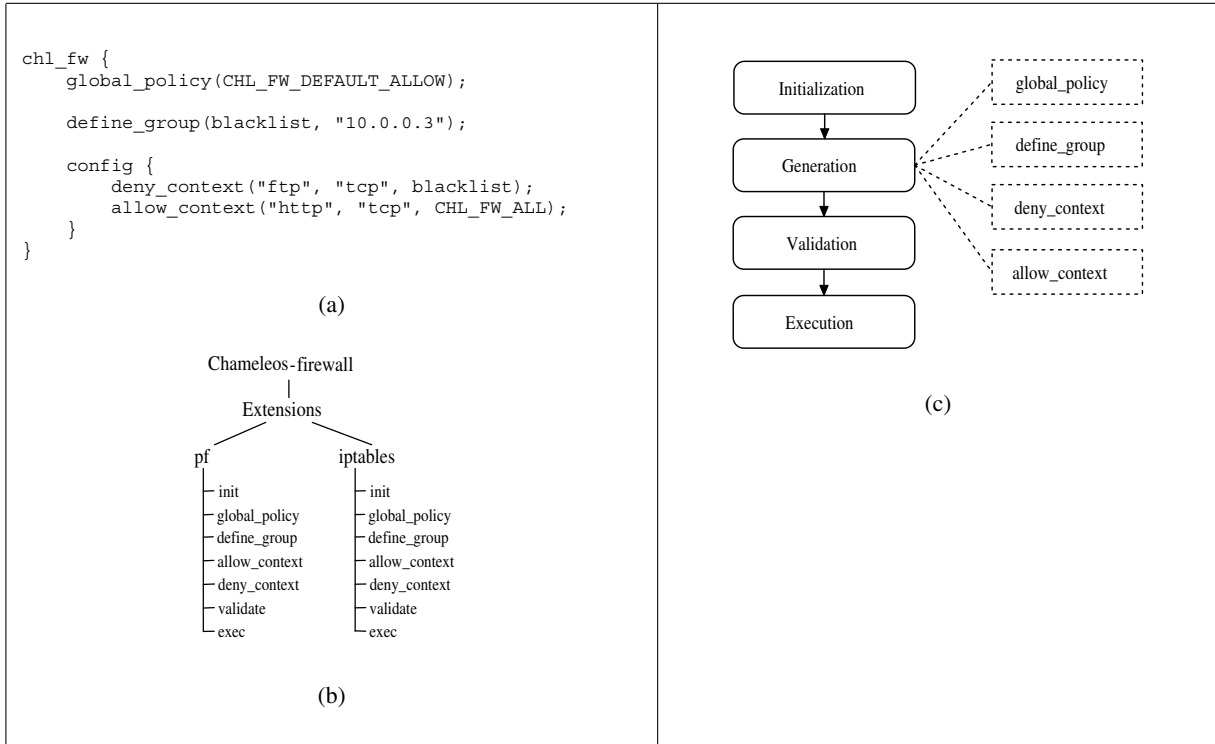tage in the translation process is gen-eration, which refers to the generation of a system policy, which is either a configuration file (as is the case for *pf*) or another type of file (like the shell script for *iptables*). If we are translating the Chameleos-`firewall` policy in Figure 2 (a), the scripts `global_policy`, `define_group`, `deny_context`, and `allow_context` would be called with the correct parameters. These scripts append the correct system-specific configuration details or commands to the system policy. The validation, as the next step, is selective since some extensions do not allow for straightforward validation. For instance, it is easy to validate a *pf* configuration file, since *pf* provides the necessary tool to do so (`pfctl -n <filename>`). However, it is not easy to validate a shell script with `iptables` commands. The last step of translation is execution. The `exec` script is called, which actually loads the system policy so that the Chameleos-`firewall` policy is enforced.

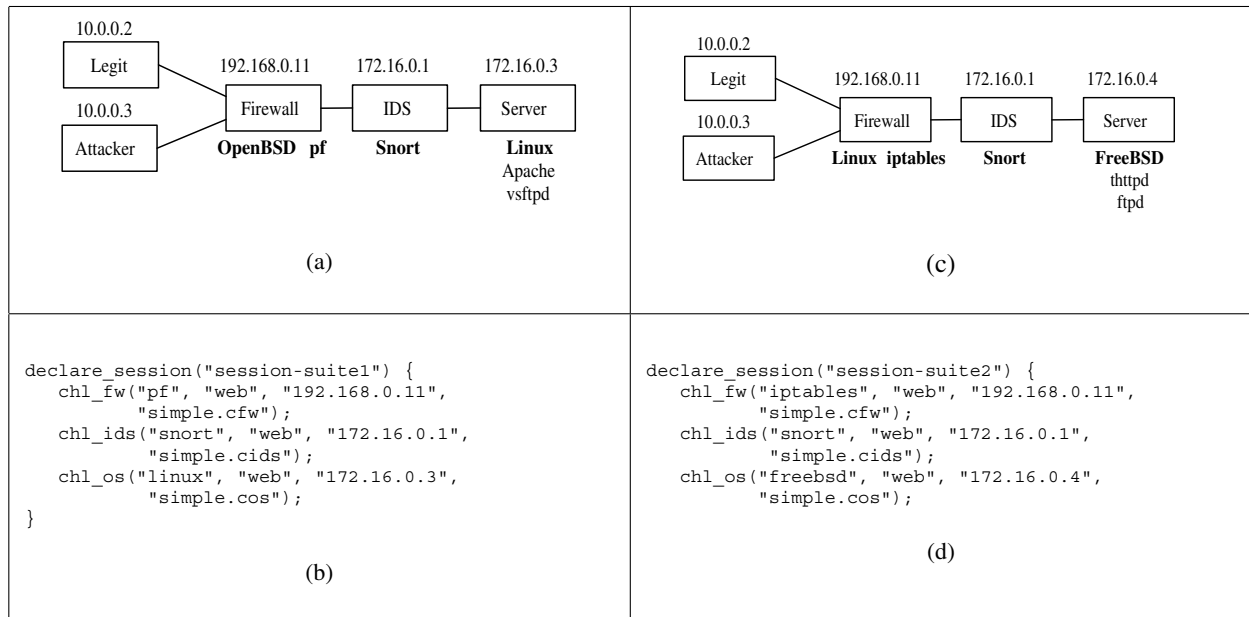## 3. Experiments and Results

Our experiments were designed with two objectives: (1) to test the translation process of each Chameleos-x variant, and (2) to test the enforcement/execution process of each Chameleos-x variant. To perform these experiments, we first designed and implemented a test network consisting of six machines, each of which plays a single role: a firewall, an IDS, a server, a "legit" machine that generates good traffic, an "attacker" machine that generates bad traffic, and the management console. Chameleos-x Enforcement Monitors were installed on the firewall, IDS, and server. The management console's responsibility is to push Chameleos-x policies to the Monitors. The Monitor is in charge of translating the Chameleos-x policy from the management console into the correct system policy for its host.

To fulfill the two experimental objectives, we use two "configuration suites" for testing each variant. A configuration suite would consist of a specific firewall, IDS, and server operating system and associated servers (such as the HTTP and FTP servers). Additionally, in order to make the experiment more accurate, we would have to make sure each suite is heterogeneous and different from the other. With that in mind, we designed the Configuration Suites as shown in Figures 3 (a) and 3 (c). Configuration Suite 1 consists of a firewall running OpenBSD with the *pf* firewalling subsystem, an IDS running Snort, and a Linux server that is geared to run Apache and vsftpd as its web and FTP servers respectively. Configuration Suite 2 comprises a Linux firewall with *iptables*, the Snort IDS, and a FreeBSD server configured to run thttpd and ftpd. We also used a management console (running NetBSD) with the IP address 172.16.0.2 to push Chameleos-x policies to the machines in each suite.

We then ran our experiment in two sessions: one with

```
chl_fw {
    global_policy(CHL_FW_DEFAULT_ALLOW);

    define_group(blacklist, "10.0.0.3");

    config {
        deny_context("ftp", "tcp", blacklist);
        allow_context("http", "tcp", CHL_FW_ALL);
    }
}
```

(a)

```
Chameleos-firewall
        |
    Extensions

pf                      iptables
  ├ init                  ├ init
  ├ global_policy         ├ global_policy
  ├ define_group          ├ define_group
  ├ allow_context         ├ allow_context
  ├ deny_context          ├ deny_context
  ├ validate              ├ validate
  └ exec                  └ exec
```

(b)

Initialization → Generation → Validation → Execution

Generation — — — global_policy
           — — — define_group
           — — — deny_context
           — — — allow_context

(c)

Figure 2. (a) A simple Chameleos-`firewall` policy - *simple.cfw*; (b) Chameleos-`firewall` extensions; (c) Translation flowchart for all Chameleos-x variants, using Chameleos-`firewall` as an example.

10.0.0.2   Legit
                        192.168.0.11   172.16.0.1   172.16.0.3
10.0.0.3   Attacker   Firewall   IDS   Server
                       OpenBSD pf   **Snort**   **Linux**
                                                        Apache
                                                        vsftpd

(a)

10.0.0.2   Legit
                        192.168.0.11   172.16.0.1   172.16.0.4
10.0.0.3   Attacker   Firewall   IDS   Server
                       Linux iptables **Snort**   **FreeBSD**
                                                        thttpd
                                                        ftpd

(c)

```
declare_session("session-suite1") {
    chl_fw("pf", "web", "192.168.0.11",
           "simple.cfw");
    chl_ids("snort", "web", "172.16.0.1",
           "simple.cids");
    chl_os("linux", "web", "172.16.0.3",
           "simple.cos");
}
```

(b)

```
declare_session("session-suite2") {
    chl_fw("iptables", "web", "192.168.0.11",
           "simple.cfw");
    chl_ids("snort", "web", "172.16.0.1",
           "simple.cids");
    chl_os("freebsd", "web", "172.16.0.4",
           "simple.cos");
}
```

(d)

Figure 3. (a) Configuration Suite 1; (b) Session file for Configuration Suite 1 – *session-suite1.chl*; (c) Configuration Suite 2; (d) Session file for Configuration Suite 2 – *session-suite2.chl*.

IEEE
COMPUTER
SOCIETY

Configuration Suite 1, and one with Configuration Suite 2. In each session, the management console sent the Chameleos-x variant policies to the respective machines on the network. We then examined the machines to see if the translation process and the enforcement/execution process worked as expected.

To run the sessions, we developed a session file for each suite (Figures 3 (b) and 3 (d)). The session file states where the Chameleos-x policy should be sent: for example, in Figure 3 (b), the `chl_fw` statement specifies that the Chameleos-`firewall` policy in the file `simple.cfw` should be sent to 192.168.0.11 and the *pf* extension should be invoked with the *web* service. We then invoked the following command at the management console:

```
$ chl-apply config session-suite1.chl
```

The `chl-apply` program sends the policies specified in the session file to the Chameleos-x Enforcement Monitors. It also has the option to specify which operation mode should be used, which in this case is `config`. This causes the Monitors to only translate the excerpt within the `config` clause in each Chameleos-x policy. The translated policies showed consistent behavior in both Configuration Suites 1 and 2, even though the same original Chameleos-x policies were used without changes in each suite. In addition, the translated policies implemented certain features using the specific facilities offered by each target system. For example, groups were defined differently in *pf* and *iptables*, but the end behavior was consistent.

These favorable results show that a practical and system-driven policy framework can be used to perform effective evaluation of a network in a flexible and extensible manner. It also firmly indicates that our policy framework could successfully integrate a simple but powerful declarative language with an enforcement architecture. The results also demonstrate that Chameleos-x, with its system- and platform-independent nature, is indeed capable of facilitating security policy management for heterogeneous environments, as represented by the consistent behavior exhibited by the multiple kinds of systems in Configuration Suites 1 and 2.

4. Conclusion

We have described the design of Chameleos-x, a practical and system-driven policy framework that can be used to facilitate the management of security policies in heterogeneous environments. Chameleos-x is designed to assist system and network developers in the configuration and evaluation of these systems for conformance to security policies. We also described our development and experimentation of Chameleos-x. Our experiments involved the feasibility assessment of Chameleos-x on our heterogeneous

"configuration suites", where each suite comprises a specific operating system, firewall, and intrusion detection system. Chameleos-x successfully demonstrated that it is able to translate a single Chameleos-x policy into the system policies for each suite, and still retain consistent behavior in each case. For the future work, we would study on new components for the Chameleos-x policy framework. Most of these new components would be part of the Chameleos-x Translator. These components include a syntax checker, analyzer, and reverse translator. The *syntax checker* would serve as the foundation for all syntax checking requirements in the other components. The *analyzer* would be used to analyze a Chameleos-x policy for conflicts and ambiguities.

References

[1] Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. A logical framework for reasoning about access control models. In *Proceedings of the 6th ACM Symposium on Access Control Models and Technologies*, pages 41–52, Chantilly, VA, 2001.

[2] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder policy specification language. In *Policies for Distributed Systems and Networks: International Workshop (POLICY 2001), Lecture Notes in Computer Science*, volume 1995, pages 18–38. Springer-Verlag, January 2001.

[3] Sushil Jajodia, Pierangela Samarati, and V.S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 31–42, Oakland, CA, May 1997.

[4] Lawrence Teo and Gail-Joon Ahn. Towards the specification of access control policies on multiple operating systems. In *Proceedings of the 5th IEEE Workshop on Information Assurance*, pages 210–217, United States Military Academy, West Point, NY, June 2004.

[5] Lawrence Teo and Gail-Joon Ahn. Managing heterogeneous network environments using an extensible policy framework. In *Proceedings of ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS'07)*, Singapore, March 2007.

[6] T.Y.C. Woo and S.S. Lam. Authorizations in distributed systems: A new approach. *Journal of Computer Science*, 6(2,3):107–136, 1993.

IEEE
COMPUTER
SOCIETY