The Case for Abstracting Security Policies

Anil Madhavapeddy[1], Alan Mycroft[2], David Scott[3], Richard Sharp[4]
University of Cambridge Computer Laboratory[1,2], Laboratory for Communication Engineering[3]
Intel Research Cambridge[4]
William Gates Building, 15 JJ Thompson Avenue, Cambridge CB3 0FD, UK
Tel: +44 (0)1223 763500 Fax: +44 (0)1223 334678
{avsm2[1],am[2],djs55[3],rws26[4]}@cl.cam.ac.uk

Abstract: As Internet connectivity grows executing untrusted code becomes an increasingly serious threat. Public Key Infrastructure (PKI) and digital signatures offer some degree of protection, but are only part of a solution. In this paper we propose a mechanism of forcing applications to "declare what they intend to do" by means of an abstract behavioural model. A monitoring process is employed to dynamically ensure that programs do not deviate from their pre-declared intention. We focus particularly on the usability, transparency and maintainability of the system, which we believe to have been lacking in similar efforts. In particular we concentrate on (*i*) building powerful and maintainable policy specification languages and; (*ii*) automatic security auditing of policies.

1 Introduction

Consider the following scenario: you are Head of Security at the Pentagon and a contractor arrives in order to carry out a specific and necessary job (e.g. building maintenance). There are clearly a number of security precautions that it is your duty to enforce. Firstly you must verify the contractor's ID. Secondly, you must assign a *minder* to watch the contractor and ensure that they only perform tasks necessary to achieving their stated goal (e.g. a plumber should not touch the computers or take documents from filing cabinets).

Now imagine the above scenario as an analogy for executing untrusted code on a modern general purpose operating system. The recent adoption of digital signature technology may serve as an ID-check, but crucially there is no equivalent of a "minder". As a result users are at the mercy of a plethora of dangerous trojans and subversion attacks. In the future the risks will continue to grow as increasing connectivity makes it easier both for unwitting users to execute untrusted code and for the applications themselves to cause damage, either maliciously (e.g. stealing credit-card numbers) or inadvertently (e.g. mishandling of input data leading to a buffer overflow attack).

In an attempt to avert this impending crisis[1] we propose a framework in which programs are supplemented with *Security Policies*. In the spirit of Schneider [12] and Hall's [7] work a Security Policy serves as an abstract model of an application, documenting its "high-level intention". A *monitoring process* (equivalent to the minder in our previous discussion) dynamically enforces the Security Policy, ensuring that the application stays true to its pre-declared intent. In order that the monitoring process can easily be retrofitted to existing general purpose operating systems and to keep the Trusted Computing Base (TCB) small we advocate the tried and tested technique of observing and restricting applications' system calls (syscalls).

A number of researchers have already implemented systems which enforce security policies via syscall monitoring [1, 2, 10, 17]. However, for a variety of reasons, none of these systems are used in practice (see Sections 2 and 4). In this paper we argue in favour of dynamic policy enforcement and address the issues which we believe are currently preventing its widespread adoption. In particular we focus on (*i*) building more powerful policy description languages which support parameterisation and code abstraction; (*ii*) using automatic model checking techniques [5] to statically check assertions against downloaded security policies; and (*iii*) automated support to assist developers in the creation and maintenance of security policies.

2 Stateful Syscall Policy Language

So far, research into dynamic Security Policy monitoring tools has focused on low-level implementation details. One of the main factors contributing to their lack of adoption is that comparatively little work has been done on developing high-level Security Policy specification languages. For example, `systrace` [10] policies essentially consist of a flat list of `allow`/`deny` rules. Policies in Uppuluri and Sekar's Intrusion Detection System [15] offer greater expressivity than `systrace` policies, but are represented as a cryptic and unmaintainable list of regular expressions over system calls. Policies in Cerb [1] and Subterfugue [3] support the validation of individual syscalls but do not provide an easy way of modeling application behaviour.

[1] In the spirit of the times, the authors reserve the right to incite fear through rampant sensationalism.

In contrast, our Stateful Syscall Policy Language (SSPL) facilitates the description of readable, maintainable and expressive policies by providing many of the features one would expect in a general purpose programming language: (*i*) an imperative programming style allows the 'state' of the program to be modeled easily; (*ii*) code sharing and policy abstraction are supported through functions, procedures and a module system; (*iii*) primitives are provided both to *validate* and *transform* individual system calls; and (*iv*) policies can be parameterised over program arguments.

Due to space constraints we cannot formally define SSPL here; instead we devote the remainder of this section to demonstrating some of its features by means of a simple example. Consider the program `ping` which sends ICMP echo packets to measure network round-trip delays and packet loss. This is an example of a program which needs "root" privileges, in a traditional Unix system, to perform a one-off special operation—acquisition of a raw socket. In such systems we are forced to make the whole program "setuid" root, trusting that it drops root privileges as soon as possible to mitigate the damage caused by an application compromise (for example the recent `ping` root exploit [11]). In our system we write an SSPL policy which describes exactly *what* syscalls `ping` makes and *when* it makes them. In the worst case scenario where an attacker successfully manages to buffer-overrun the program, the system prevents the compromised `ping` binary from violating the policy.

SSPL policies are equivalent to non-deterministic finite state automata, specified in a convenient 'C'-like syntax. A policy in SSPL looks superficially like a vastly simplified version of the program's source code, with the majority of the application logic removed.

For any particular application there is a large set of possible policies that can be written (ranging from simple, coarse-grained policies through to more comprehensive models of the program behaviour). Although there is room for debate about the optimal granularity of Security Policies (e.g. how much of the application does one model?) the key observation is that policies are invariably several orders of magnitude less complex than the corresponding applications. Our intention is that the Security Policies are sufficiently simple to facilitate automatic model-checking against a set of global system security rules and, in some extreme cases, even manual inspection (see Section 3).

In the face of incomplete information about application state, SSPL allows conditional branches to be modeled by a non-deterministic choice construct written "`either/or`" (cf. Occam's `ALT`). The `multiple` keyword, similar to the repetition construct in regular languages, signifies that a block of statements may be repeated zero or more times.

At runtime the application's observed behaviour (i.e. the sequence of syscalls invoked) is compared with the SSPL policy. Any attempt by the application to make an unexpected syscall results in immediate termination. The core of a (fairly fine-grained) OpenBSD policy for `ping` is as follows:

```
import libc;

function ping(stdout, sock){
   sendto(sock, *, *, *, *, *);
   recvfrom(sock, *, *, *, *, *);
   optional libc.printf(stdout);
}

function main(stdin, stdout, stderr, host) =
  always-allow ( break, mprotect ) {
    either {
      // Showing program usage
      libc.printf(stdout);
      libc.exit(*);
    } or {
      // this block makes use of async signal handling
      during {
        // Perform some pings
        let ping_sock = socket(AF_INET, SOCK_RAW, *);

        always-allow ( libc.gettimeofday ) {
          // if supplied a hostname we must resolve it
          optional libc.gethostbyname( host );

          // print out the headers
          libc.printf(stdout);

          // Set up a bunch of signal handlers
          multiple { sigaction(*, *, *); }

          // perform some number of pings
          multiple { ping(stdout, ping_sock); }
       } handle {
          // on receipt of a signal (e.g. SIGTERM)
          // print results and exit
          libc.printf(stdout);
          libc.exit(*); }
     }}}
```

As can be seen from the code fragment, policies may be divided into a number of `functions` and factored into libraries via the `import` keyword. System calls (in the above example we use `sendto`, `recvfrom`, `break`, `mprotect`, `socket`, `gettimeofday` and `sigaction`) are modeled by functions which take the same number of formal parameters as the real syscalls. In the policy domain, function arguments range over constants, variables or wild-card patterns (e.g. '`*`'). At runtime, if the arguments of the syscall do not match the argument patterns specified in the policy then the application is terminated.

We assume the existence of a library of policies corresponding to functions contained within the standard Unix libraries, e.g. in the example above we assume the existence of functions `libc.printf` and `libc.gethostbyname` for printing output and performing DNS lookups respectively. Our example highlights the importance of abstraction in a policy description language. By mirroring the Unix library calls in our policies we significantly enhance the readability and maintainability of SSPL specifications.

Within a policy variables may be bound to function arguments and the return values of system calls. In the example above, the variable `ping_sock` is bound to the raw socket file descriptor returned by the call to `socket` and passed to the function `ping` which allows datagrams to be sent only to this particular socket.

Transformations (not present in the above example) are possible using the `transform` keyword as in the following mini-example:

```
transform open(filename, flags, mode) ->
   open("/chroot/" + filename, flags, mode)
```

The special function called `main` represents the entry point of the specification. Through this function, the policy is instantiated with the same set of arguments as the application itself. Parameterisation over program arguments facilitates the specialisation of a policy to a particular *invocation* of a program at load time. For example, consider the program `cp` which copies a file. Without knowledge of the arguments being passed the policy would have to allow the program access to *all* files on the system. However, if the policy knows the arguments being passed then it can restrict `cp` to touch only those files passed as arguments. At first sight there appears to be a need for a trusted shell which promises to pass the same arguments to both policy and program. Happily this is not the case: by protecting the shell with its own Security Policy that intercepts `execve` syscalls the responsibility of passing arguments to both application and policy becomes the duty of our trusted *monitor process* (see Section 3). This is an important point as it keeps the TCB small.

Some application behaviour depends on the exact runtime environment of the program: for example, the order of signal handling calls depends on the behaviour of the system scheduler and the IO access patterns of a program depend on the disk scheduler. To avoid cluttering the specification we provide two useful constructs: `always-allow` and `during/handle`. The former takes a list of system calls which are ignored in the subsequent statement block. In the above example this feature is used to allow unrestricted memory management through calls to `break` and `mprotect` for the entire duration of the program. The "`during/handle`" construct specifies that the policy block marked by "`handle`" may occur at any point within the block marked "`during`". In the above example we use this feature to represent syscalls invoked from an async signal handler in response to the user hitting Control+C. In general the construct is useful for modeling resumable exceptions.

SSPL encourages the creation of *evolveable* policies. A policy can be progressively refined to specify the behaviour of the program in increasing levels of detail. One useful refinement is to attach a network protocol analyser (such as SPECTRE [13]) to reconstruct messages being sent on the wire by calls to the `write` syscall. In the `ping` example this technique would allow us to ensure that all packets sent by the program are actually valid ICMP echo packets and do not, for example, contain data designed to attack another machine.

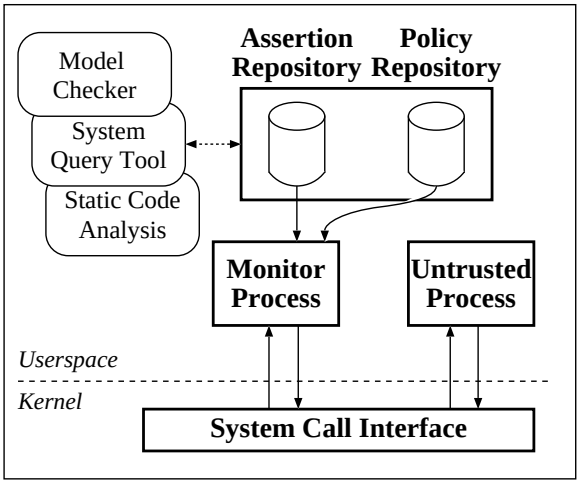

Figure 1: Dynamically enforcing security policies using a monitoring process

3 System Structure

Figure 1 illustrates the core components of the system: (*i*) a monitoring process which dynamically enforces the Security Policies; (*ii*) a kernel interface which intercepts system calls of interest to the monitoring process; and (*iii*) a secure repository which stores both global assertions and process' security policies (see below). We use the monitoring process itself to maintain policy integrity through a global assertion which prohibits any processes from accessing the repository. Of course, since the monitoring process is part of the TCB and isn't itself monitored, it is able to access the repository.

All processes (except the monitor process) must have an associated Security Policy. Since it is unrealistic to assume that every application will have its own policy in the foreseeable future, we provide a set of generic template policies which reside on the client machine. These generic policies can be used to constrain the behaviour of any application. By default, the system chooses the most secure template, but gives the user the option of upgrading the application's security level if required (for example, for external network access). We acknowledge that the use of generic policies in this way is not ideal, but we emphasise that it is still better than nothing. Perhaps most importantly it provides a way in which our system can be deployed usefully before the number of applications with associated security policies reaches a critical mass.

We also envisage a number of other components which, whilst not essential, greatly aid in the manageability of the system. Following the work of Wagner and Dean [16], we intend to use Static Code Analysis to assist policy creators in cases where application source-code is available. However, whereas Wagner and Dean generate policies represented as (large) state machines, the policies we generate

will be expressed in the more readable SSPL. The difficult part of automatic policy generation is to choose the right-level of policy granularity. Whilst it is unrealistic to expect the quality of automatically generated policies to be as good as manually constructed policies, our hope is that they will still be better than nothing. By integrating static analysis tools into build infrastructure, such as `autoconf` and `automake`, one could generate policies automatically at compile-time for a large set of existing applications.

3.1 Auditing Policies

So far we have discussed a mechanism for dynamically checking applications against their associated security policies. Let us now consider what happens when a user downloads a new untrusted application which comes with its own Security Policy specification. Of course, the policy itself cannot be trusted: a cracker may simply have written a policy to accompany their latest piece of malware!

The one thing that we can be sure of is that an application will only perform actions that are allowed by its associated policy (since this is enforced by our monitor process which is part of the TCB). Thus, if we can determine that the policy is safe to run, we can infer that the application itself is safe to run. The key is that, since we intend policies to be several orders of magnitude simpler than applications (see Section 2), we have similarly reduced the complexity of the security auditing process by several orders of magnitude.

Researchers have already investigated the automatic security auditing of applications using a variety of techniques including code analysis and proof checking [9]. We strongly believe that this work will be far more applicable if one only has to solve the much simpler problem of auditing abstract policy specifications.

We intend to use model checking to statically prove *security auditing assertions* against an application's Security Policy, before it is executed. As shown in Figure 1 our system has an *assertion repository* which contains the requirements that a Security Policy must satisfy before its associated application is deemed acceptable for execution. In accordance with current work on model checking, these rules would be specified in one of the existing temporal logics (e.g. CTL or LTL). There are many ways in which the assertion repository could be organised. For example, some assertions may be global (checked against all policies) and other assertions may be predicated on the system's current security level (e.g. low, medium, high).

We extend this notion of assertion checking via a tool which allows users to execute high-level queries against the policy repository (e.g. is there a program which may write to the network after reading from "My Documents"?). Although expert users may be able to pose such queries in formalisms such as Modal Logic [4], we envisage a standard set of common queries for novice users.

4 Discussion

There are several ways in which our Security Policy specification and enforcement framework can be used. Firstly, as we have already argued, it provides the function of a "minder", forcing untrusted code both to declare its intentions and keep to its word. However, another angle is that the tools can be put to good use in the software development process. Research has shown that manually coding to limit the impact of subversion attacks (e.g. buffer overflows) is a time-consuming and complicated process.

One of the major benefits of our policy enforcement framework is that the Trusted Computing Base (TCB) remains small (consisting only of code to dynamically monitor syscall traces and enforce security policies). Furthermore, the vast majority of the integrity checking occurs in userspace; the only kernel modification necessary is the addition of a small module to intercept syscalls (see Figure 1). Previous research has shown that this design methodology does not impose significant performance overhead [10].

An interesting observation is that we can use our Security Policy specifications to implement a variety of existing security mechanisms. For example, without increasing the size of the TCB, we can use our syscall `transformation` rules to subsume systems such as `chroot` and `jail`. Also note that if one is prepared to augment the TCB with a mechanism for storing persistent data we can subsume more advanced mechanisms using SSPL such as the Mandatory Access Control system in TrustedBSD [17] or SE Linux [8].

There are a number of parallels between our approach and that of Proof Carrying Code (PCC) [9]. One of the major barriers to adoption of PCC is the difficulties that developers face in generating proofs of security properties. Our approach offers a potential solution to this problem: since we provide much simpler abstract models of applications, one only has to generate proofs of properties against these models. Although our primary intention is to use model-checking to verify properties on the client-side, we could also choose to explore a more PCC-like approach. In this case client-side proof checkers verify pre-generated proofs against applications' Security Policies. Since the abstract models are several orders of magnitude less complex than the applications themselves, the problem of proof generation is simplified considerably. Our approach offers a means of smoothly moving the burden of proof along a static/dynamic verification spectrum.

Capability systems are becoming increasingly integrated into existing general purpose operating systems. Whilst these provide more fine-grained security than the traditional Unix model, they suffer from a number of deficiencies that our framework addresses. Capabilities (*i*) only protect a fixed set of privileged operations (e.g. the ability to bind to a low port); (*ii*) require modification of application source; and (*iii*) trust applications to use the capabilities in a secure manner (i.e. not to hold onto high-privilege capabilities unnecessarily). In contrast our ap-

proach alleviates these problems: since we use a programmatic framework we can write policies of arbitrary granularity; application source code does not need to be modified; and we no longer trust applications (since we force them adhere to their policies). Additionally, we open the possibility of auditing security policies automatically.

5 Conclusions and Future Work

The existing "suck it and see" model for dealing with untrusted code is clearly inappropriate. Whilst untrusted code has always been a problem, increasing connectivity and an increasing reliance on computers for financial transactions threaten to make the problem an even more serious threat in the near future. Although a number of researchers have demonstrated that dynamic Security Policy enforcement through system call monitoring may offer an effective [15] and low-overhead [10] security layer, there are a great many problems which must be addressed before such techniques can be widely deployed. The purpose of this paper is to highlight these problems which as yet remain largely unsolved, argue for their importance and outline some of the preliminary research we have done in this area.

We acknowledge that system-call tracing is insufficient to guard against certain classes of attacks, for example "mimicry" attacks [6]. We hope to enhance our system in a number of ways including (*i*) validating all I/O by filtering through protocol analysers [15]; and (*ii*) using dynamic software translation techniques [14] to more tightly integrate the policy with the application binary.

"Untrusted code" is just as much a social problem as it is a technical problem. Looking for a *complete solution* is unrealistic: it is analogous to looking for a solution to crime in general. With this in mind, we do not claim that our proposed framework is a panacea. However, although a number of security problems remain (e.g. covert channel leakage), we claim that our system offers the potential to raise the security level of existing general purpose operating systems significantly.

6 Acknowledgments

This work was supported by Network Appliance, Inc., Intel Research, AT&T Laboratories Cambridge and the Schiff Foundation. The authors wish to thank Steven Hand, Ian Pratt, Tim Harris, Tim Deegan, Derek Mcauley, Alastair Beresford and Andrew Moore for their valuable suggestions.

References

[1] Cerb. `http://cerber.sourceforge.net`.

[2] Janus: Sandboxing untrusted applications. `http://www.cs.berkeley.edu/~daw/janus`.

[3] Subterfugue: Playing with the reality of software. `http://www.subterfugue.org/`.

[4] L. Cardelli and A. D. Gordon. Anytime, Anywhere Modal Logics for Mobile Ambients. In *Principles of Programming Languages (POPL)*, 2000.

[5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. Cambridge, MA: MIT Press, 1999.

[6] D.Wagner and P.Soto. Mimicry attacks on host based intrusion detection systems. In *Ninth ACM Conference on Computer and Communications Security*, 2002.

[7] R. Hall. Open modeling in multi-stakeholder distributed systems: requirements engineering for the 21st century. In *Proceedings of the First Workshop on the State of the Art in Automated Software Engineering*, June 2002.

[8] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Freenix Track of Usenix Annual Technical Conference*, 2001.

[9] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, Jan. 1997.

[10] N. Provos. Improving host security with system call policies. Technical report, Center for Information Technology Integration, University of Michigan, November 2002.

[11] Red Hat, Inc. Security Advisory RHSA-2000:087-02. Ping Buffer Overflows.

[12] F. B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.

[13] D. Scott and R. Sharp. Abstracting Application-Level Web Security. In *The Eleventh International World Wide Web Conference Proceedings*, pages 396–407, May 2002.

[14] K. Scott, J. Davidson, and K. Skadron. Low-overhead software dynamic translation. Technical Report CS-2001-18, 2001.

[15] P. Uppuluri and R. Sekar. Experiences with specification-based intrusion detection. *Lecture Notes in Computer Science*, 2212, 2001.

[16] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, 2001.

[17] R. Watson. TrustedBSD: Adding trusted operating system features to FreeBSD. In *USENIX Technical Conference*, 2001.