SubDomain: Parsimonious Server Security

Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle and Virgil Gligor[1]
– WireX Communications, Inc.

ABSTRACT

Internet security incidents have shown that while network cryptography tools like SSL are valuable to Internet service, the hard problem is to protect the server itself from attack. The host security problem is important because attackers know to attack the weakest link, which is vulnerable servers. The problem is hard because securing a server requires securing every piece of software on the server that the attacker can access, which can be a very large set of software for a sophisticated server. Sophisticated security architectures that protect against this class of problem exist, but because they are either complex, expensive, or incompatible with existing application software, most Internet server operators have not chosen to use them.

This paper presents SubDomain: an OS extension designed to provide sufficient security to prevent vulnerability rot in Internet server platforms, and yet simple enough to minimize the performance, administrative, and implementation costs. SubDomain does this by providing a *least privilege* mechanism for *programs* rather than for users. By orienting itself to programs rather than users, SubDomain simplifies the security administrator's task of securing the server.

This paper describes the problem space of securing Internet servers, and presents the SubDomain solution to this problem. We describe the design, implementation, and operation of SubDomain, and provide working examples and performance metrics for services such as HTTP, SMTP, POP, and DNS protected with SubDomain.

Introduction

Common server operating systems such as Linux, Windows, Solaris, etc. are subject to vulnerability rot as security vulnerabilities (i.e., implementation bugs) are discovered in the component software of these operating systems. For instance, a buffer overflow discovered in the BIND domain name server [15] allowed remote attackers to gain root privileges on a variety of system platforms, and a similar vulnerability in Microsoft's IIS (web server) [21] allows remote attackers to gain administrative control of Windows servers. The recommended defense for general purpose servers is to keep the host system up to date with vendor patches to close these vulnerabilities.

However, many of these systems are being pressed into use as the basis for *server appliances*: servers intended for largely unattended operation by unskilled users. But because these operating systems are subject to vulnerability rot, they need to be frequently upgraded with vendor patches. While this is an acceptable approach for general purpose servers (where a skilled system administrator is expected to maintain the system) it is not acceptable to appliance users, who expect a device with the maintenance factor of a toaster.

[1]Cowan, Beattie, and Pu are formerly of the Oregon Graduate Institute, where much of this work was done. Cowan and Beattie are now with WireX Communications, and Pu is now with the Georgia Institute of Technology. Wagle is still with the Oregon Graduate Institute, and Gligor is with the University of Maryland. Kroah-Hartman is with WireX.

The classical security solution to vulnerability rot is the notion of *least privilege*: the technique of granting subjects in a system precisely the capabilities they need to perform their function, and no more [33]. Effective use of least privilege *minimizes* the potential damage that results when a trusted program is penetrated by minimizing the degree to which the program is trusted.

Security architectures that provide least privilege mechanisms exist, but because they are either complex, expensive, or incompatible with existing application software, appliance vendors have not chosen to use them. Existing defenses entail these complexities precisely because they were designed to handle the generality of a general purpose server, and thus must deal with user least privilege. This generality complicates the least privilege abstraction, making the enforcement mechanism more complex to implement and use.

This paper present SubDomain: an OS extension designed to provide sufficient security to prevent vulnerability rot in server appliances, and yet simplify as much as possible to minimize the performance, administrative, and implementation costs. SubDomain does this by providing a least privilege mechanism for *programs* rather than for *users*. The security restrictions *complement* the system's existing permissions, allowing a program to be secured independent of who may be using the program. This notion is especially effective on server appliances, and enables program-specific confinement information to be distributed *with the program* (see the section on SubDomain compatibility).

By specifically addressing least privilege for programs, we can provide a mechanism that has a relatively small implementation and simple operation. Small implementations are important for security systems to avoid vulnerabilities due to bugs in the enforcement mechanism itself. Simple operation is important for security systems to avoid misconfiguration. Even more so than in most OS design issues, parsimony is critical to security [33] making SubDomain's relative simplicity of design and implementation an important feature.

We present the SubDomain notation for recursively specifying the sub-domain of resources available to a software component, our implementation of SubDomain as an enhancement to the Linux kernel, our application of SubDomain confinement to several example applications, performance metrics on the cost of SubDomain confinement, and our analysis of the security of a system protected by SubDomain.

The challenge of supporting least privilege is to provide a specification system that is expressive enough to specify privileges that are actually minimal, is convenient enough that administrators can reasonably specify least privileges, and yet preserves compatibility and performance. While SubDomain strives for simplicity relative to other least privilege mechanisms, it provides for finer granularity least privilege in one important regard: SubDomain can confine *arbitrary* software components, at a finer granularity than the native OS process, i.e., procedures and modules. This is especially important for component-based services such as Apache [9] and its loadable modules (see the section on confinement).

The rest of this paper is organized as follows. The next elaborates on the problem of vulnerable/buggy software, and describes the abstract solution of *least privilege* to minimize the potential damage due to attacks against vulnerable software. Readers familiar with least privilege can skip ahead to the third section, which describes the SubDomain security enhancement, and how it advances over previous least privilege mechanisms by providing finer granularity, and simplifying the problem of confining suspect *programs*. The fourth demonstrates SubDomain's compatibility by confining assorted software components, including *sub-process modules*. The fifth section presents the performance costs of SubDomain confinement. The subsequent section 6 describes related work specifically addressing the problem of confining suspect programs. The final section presents our conclusions.

The Problem: Vulnerable Programs and Least Privilege

Many security vulnerabilities result from bugs in "trusted" programs. A "trusted program" is a program that runs with privilege that some attacker would like to have, and the program fails to keep that trust if there is a bug in the program that allows the attacker to acquire that privilege. Some examples include:

Buffer Overflows: Many privileged programs contain "buffer overflow" vulnerabilities, a problem endemic to C programs that provide poor bounds checking on user-supplied input. Buffer overflows are very common [18, 19] and very dangerous [32, 29], allowing attackers to take control of programs from an anonymous node on the internet.

Race Conditions: Many privileged programs also contain "race condition" vulnerabilities. Here, the problem is that careless root privileged processes create files without adequate checking for the prior existence of the file. The problem is that the attacker can create a symbolic or hard link in the file system between the time the privileged program checks for existence and the time it creates the file, with the result that the root program unwittingly uses its authority to corrupt some other critical file [12].

Special Character Processing: While few root privileged programs are written in shell scripting languages, many other programs with "interesting" privileges are written as shell scripts, especially CGI/ PERL [41] programs for processing web forms. CGI programs run with the authority of the web server, and must process arbitrary input from arbitrary users. If the attacker can provide input (using creative URLs) to a CGI program that yields control to the attacker, then the attacker can gain control of the web server, e.g., the PHF program (included in early NCSA and Apache web servers) allowed the attacker to present a URL to the web server that would cause PHF to start an `xterm` on the attacker's display [14].

Note that while "trusted" usually refers to highly privileged processes (e.g., root processes) they can actually be processes with *any* privileges that the attacker wants but does not have. The general case is that any program installed on a computer that processes input from potentially hostile users becomes a potential vulnerability. Eliminating these vulnerabilities requires some form of assurance that the program in question does not contain exploitable bugs, but this kind of assurance is problematic. Some classes of bugs, e.g., buffer overflow vulnerabilities, can be eliminated through various compiler techniques [39, 17, 26, 37]. Other forms of vulnerabilities are undetectable at compile time, e.g., race conditions [12] and general logic errors.

The only way to assure the complete absence of a security vulnerability in a program is through expensive manual verification. In the absence of such verification, one must either suffer the risk of potential vulnerabilities, or *contain* the potential damage. Note that the activities we seek to constrain are "those that cause damage to the system," i.e., *safety properties* [1] with respect to *integrity*. We are not addressing other security issues, such as *information flows* [27] that might disclose secrets. Readers already familiar with least privilege mechanisms can skip to to the next section for a description of SubDomain, our contribution to the field.

The Solution: Least Privilege

The classic solution to the problem of unknown security vulnerabilities is to perform each activity with the *least privilege* required to complete that task [33]. While this does not stop exploitation of these vulnerabilities, it does contain the damage as much as possible. An attacker who gets control of a least privilege process can, at most, read secrets and corrupt data that the exploited process has access to, and no more.

The challenge of supporting least privilege is to provide a sufficiently *fine-grained* mechanism to specify privileges that are actually minimal, while also preserving compatibility and performance. It is conceptually simple to divide system privileges into fine-grained units and then attribute the exact required privileges to a given activity, but the result of such an approach is specification notation that is tedious to maintain (breaking compatibility) and an enforcement mechanism that is slow (breaking performance).

Practical least privilege therefore involves abstracting the system resources to expedite least privilege specifications. Matching least privilege abstractions to native OS resources in turn enables efficient least privilege enforcement. Least privilege is also a useful notion in managing *user* privileges, leading many systems to combine least privilege for users and programs into a single mechanism, as described in the next subsection.

However, if the problem is bugs in programs that can be accessed by completely untrusted users, then user-oriented least privilege mechanisms may become awkward or inadequately expressive. The section on users and rolls describes some more elegant approaches to using user privilege mechanisms to confine suspect programs. The third section discusses SubDomain, our OS security enhancement that particularly address the problem of least privilege for programs, and a penultimate section discusses related work specifically aimed at program confinement.

Using User Privileges to Confine Programs

Least privilege for users is a classic way of structuring a system, and many operating systems provide facilities for constraining the privileges of a given user. User-oriented least privilege facilities can be adapted to confining collection of programs by creating a *synthetic* user, and then running the program as that user.

The classic example is the UNIX setuid facility: the `setuid` bit for an executable file indicates that the program runs with the privilege of the owner of the file instead of the privilege of the invoking user. Often this is used to create `setuid root` programs that provide controlled access to protected resources by expanding the privileges the program runs with to be all of root's privileges. To use setuid to confine a program to a smaller set of resources, a new synthetic user can be created that has those privileges, e.g., `nobody`. Programs can then be made `setuid nobody` to confine their actions to a small set of privileges.

One limitation to this approach is that all user-IDs, even synthetic user-IDs, can access all files on the system that permit "other" accesses. Another limitation to this approach is that only root can create new user-IDs. The result is that normal users cannot construct ad hoc "sandboxes" for programs that they may choose to install and run. Users are then left with their choice of:

- beg the system administrator to create a new user-ID for them,
- do not install software that is not trusted,
- run untrusted software without protection, none of which is very appealing.

So in principle, synthetic user-IDs and the `setuid` mechanism can support least privilege for programs, but in practice it forces root to do all the work. Therefore this technique is rarely deployed, people run untrustworthy software with much more privilege than is necessary, and suffer the consequent security risks.

Users and Roles

Because synthesizing user-IDs is awkward, the notion of a *role* emerged. A role is a collection of related privileges [2]. In 1986, Bobert and Kain introduced the notion of *type enforcement*: objects (files) are assigned to *types*, subjects (processes) are assigned to domains, and tables determine which domains have access to which types. Badger et al expanded on this notion [7, 8]. In a similar vein, *role-based access control* (RBAC) [22, 34] assigns users to roles, and then grants privileges to the roles.

Similar to the `setuid` approach described previously, roles can be pressed into service confining programs to a least privilege set of resources by assuming a specific role just prior to executing the program. While using roles to confine programs is more elegant than synthesizing user-IDs, it is still fundamentally overloading a user-oriented access control mechanism to manage software defects. In the next section, we describe our mechanism to specifically address the problem of vulnerable software.

SubDomain Security: Recursive Component Confinement

SubDomain is a kernel extension designed specifically to provide least privilege confinement to suspect programs. SubDomain allows the administrator to specify the *domain* of activities the program can perform by listing the files the program may access, and the operations the program may perform.

SubDomain restrictions *complement* the native access controls, in that SubDomain never *expands* the set of files a program may access, i.e., any file access must pass the native access controls and the SubDomain restrictions before access is granted. Thus SubDomain confinement makes a program monotonically safer to run.

The next subsection describes the SubDomain notation and semantics. Then, the subsequent subsection explains how SubDomain leverages work in safe programming models like *proof-carrying code* [30] to achieve component confinement below the granularity of a native process. The final subsection describes the SubDomain implementation.

SubDomain Notation & Semantics

Figure 1 shows a trivial SubDomain specification, in which the foo program is given read access to the /etc/readme file, write access to the /etc/writeme file, and execute access to the /usr/bin/bar file. When ever the program foo is run, by any user, it is restricted to access these specified files with these modes. SubDomain profiles can also grant access to directories through simple globbing, i.e., the profile in Figure 1 grants the foo program to all files in /mydir.

```
foo {
 /etc/readme       r ,
 /etc/writeme      w ,
 /usr/bin/bar      x ,
 /mydir/*          r ,
}
```

Figure 1: Trivial subDomain.

```
foo {
 /etc/readme   r ,
 /etc/writeme  w ,
 /usr/bin/bar  x +{/etc/otherwrite w} ,
 /usr/bin/baz  x -{/etc/writeme    w} ,
}
```

Figure 2: Relative SubDomain

The x (execute) capability is of particular importance: what restrictions should apply to the child process? By default, the child process inherits the parent's SubDomain, preventing the confined program from "escaping" its confinement by executing an unrestricted child process. However, sub-components of an application may require different capability sets than the application as a whole. For instance, games only need strong privileges to initialize video controllers, and mail delivery agents only need strong privileges to actually write to a user's mail box. Thus child programs can be given different constraints by specifying a *relative subdomain*, denoted by a x followed by a + or - followed by a SubDomain specification. For example, Figure 2 shows a SubDomain for foo that says that when the sub-component bar is run, it can *also* have write permission to the /etc/otherwrite file. Conversely, it says that when foo runs the sub-component baz, it may not write to the /etc/writeme file.

Sub-components may also want a SubDomain that is completely unrelated to the parent domain. For example, a web server application might need to send some e-mail while processing a web form, and thus invokes a mail delivery agent whose SubDomain is completely different. We support this need with *absolute subdomains*, denoted by a subdomain specification following an x without a + or a -. Figure 3 shows an example absolute subdomain in which the bar program run from the foo program has access to a completely different subdomain than the foo program.

```
foo {
    /etc/readme          r ,
    /etc/writeme         w ,
    /usr/bin/bar         x  {
        /usr/lib/otherread   r ,
        /var/opt/otherwrite  w ,
    } ,
}
```

Figure 3: Absolute SubDomain.

When a confined process tries to perform a file operation that is not permitted, two things happen:

1. The syscall returns with the error EPERM, just as if the attempt had failed due to a standard UNIX file system permission error.
2. The kernel generates a syslog entry describing the attempted violation. Intrusion detection systems can thus collect what ever information they want, and act accordingly, e.g., kill the offending process if such drastic steps are desired.

Sub-process Confinement

The section on related work describes several other systems that provide program-confinement mechanisms. However, with the exception of Java [4] the smallest component that they can confine is a native OS process. In contrast, SubDomain provides the unique feature of being able to confine components that are only a portion of an OS process. Historically of little practical interest, the need for sub-process confinement comes from the rise in popularity of scriptable servers and loadable modules. Let us expand upon these concepts.

A "scriptable server" is a server program that, from time to time, interprets a script or a program within itself, i.e., server-side includes [5], PHP web pages [6], Java servlets [3], etc. Such scripts are legitimately sub-component programs requiring separate confinement. Scriptable servers often have their own security mechanisms, but in depending on such restrictions, we are depending on application correctness, which is the dependence we seek to avoid in the first place. We would rather have a confinement mechanism that can be enforced by the operating system so that we do not depend on the correctness of the server application.

"Loadable modules" or "plug-ins" is the notion of providing a (fairly) fixed API in an application so that extensions to can be loaded into the application, either at start-time or run-time. "Plug-in" is the common term for desktop applications (i.e., Netscape Navigator & Shockwave, Microsoft Word and EndNote) while "module" is the common term for servers (e.g., Apache and mod_perl).

The mod_perl module for Apache provides a perfect example of the sub-process problem. PERL scripts run at the behest of the Apache web server are normally interpreted by starting a separate process to run the PERL interpreter, and then interpreting the PERL script in that separate process. mod_perl loads a PERL interpreter directly into the Apache process to avoid the cost of starting the PERL interpreter process. While this is good for server throughput, it is bad for security:

- Bugs in mod_perl can crash the Apache web server process.
- Program-confinement mechanisms that only operate on OS processes cannot confine scripts interpreted by mod_perl separate from the Apache web server process.

The SubDomain solution to the "scripting & module" problem is to provide for subdomains for sub-process components, in cooperation with the enclosing application. The notation for a sub-process subdomain is unchanged from that of separate-process subdomains shown in Figure 1 through Figure 3. The effect is to create a variety of "hats" that process can wear, one for each sub-process component that it calls. The "cooperation" required from the enclosing program is that it should call the new change_hat() system call before calling the sub-process component.

The requirement to call change_hat() implies that we are once again trusting the application, which SubDomain is supposed to avoid. However, we are trusting the application code a great deal *less*, in that the application only has to make appropriate calls to change_hat(), which is much simpler than constructing and enforcing an effective "sandbox" environment [20]. Successfully calling change_hat() with the name of a sub-component before calling the sub-component seems easy enough to do correctly.

In addition to the requirement that enclosing application correctly calls change_hat(), we also require that the sub-component does *not* call change_hat() to escape to a more liberal subdomain. Here, we employ a *cookie* argument to change_hat() to prevent the confined module from escaping. The containing process initially calls change_hat() with a particular cookie value, and further change_hat() calls that do not provide a matching cookie argument are treated as security violations.

Thus for the containing process to prevent subcomponent from escaping from the change_hat() SubDomain, it need only provide a cookie value that the contained sub-component cannot easily guess. We recommend fetching a word from /dev/random, but any reasonable source of entropy can be used.

The security of this method depends on the subcomponent *not* being able to read the parent process's cookie value. Here, SubDomain can leverage the power of *language*-based security protection systems such as proof-carrying code [30], strong type checking [39, 26, 37], and other language-based protection schemes [24, 40]. Such methods can, in principle, *prove* that the sub-component will not invoke the change_hat() system call.

Programming language techniques provide powerful protection, but also impose significant practical constraints, not the least of which is that the sub-component needs to be written in a particular language. In practice, we can still get reasonable assurance that the sub-component cannot read the containing process's cookie value if it is written in a scripting language, i.e., a language that is *interpreted* rather than one compiling to native CPU instructions. In the practical setting of scripts for web servers, most such programs that are executed by loadable modules are scripting languages, e.g., PERL [41], PHP [6], and Java [3]. While no *formal* assurances are available, in practice it is easy to trust, say, mod_perl to not address random memory.

To see the power of this approach, consider the chronic problem of securely supporting Microsoft's "Front Page Extensions," a collection of non-standard HTML tags that the server interprets to provide more dynamic HTML content. Microsoft provides a mod_fp Apache module and collection of helper programs, but they have a poor security history [36]. There is no current practical method to securely support mod_fp.

SubDomain can solve the mod_fp problem by treating web pages containing "Front Page Extensions" tags as sub-components, and assigning each such page to a subdomain. So long as the mod_fp module can be trusted not to call the change_hat() system call, then no errant action of mod_fp can violate the security policy of the subdomain for the page it is interpreting.

SubDomain Implementation

The basic architecture of SubDomain is shown in Figure 4. The SubDomain policy engine is implemented as a Linux [38] loadable kernel module. Following the usual UNIX permissions checking, the relevant system calls (open(), exec(), read(), etc.) are modified to check if the calling process is a confined process. If so, the request is referred to the SubDomain module for further inspection. The SubDomain module then either returns normally (if the request is permitted) or returns an EPERM error (if the request is denied).

Once loaded, the SubDomain module disables module unloading to prevent tampering with the SubDomain policy engine. A user-level parser reads subdomain profiles from /etc/subdomain.d/* to convert the textual representation of profiles into kernel data structures, and inserts the updated profiles into the kernel via a sysctl() interface. By convention, the /etc/subdomain.d/foo file would confine the foo program, but as shown previously, the actual name of the confined program is in the file, so confining multiple components with a single file is possible. Only root processes can access this kernel interface, and SubDomain-confined programs may not *access* the profile interface. In future work we will add further authentication requirements to the kernel's profile interface.

SubDomain Parsimony

SubDomain is simpler than competing least privilege mechanisms described in the section on related work in both implementation and usage. With regard to implementation, the SubDomain module and kernel patches amount to 4500 lines of C code, and the non-kernel parser is 825 lines. In contrast, the DTE kernel enhancement [7, 8] is over 40,000 lines of code. The relatively simple semantics of SubDomain enable a smaller implementation. "Smaller" is important for security systems, where correctness is critical, because bugs are approximately liner in code size.

SubDomain's usage is simpler than its competitors in that it is easier to devise and inspect SubDomain confinement profile than in other systems, which we elaborate on in the next section.

SubDomain Compatibility

We test the compatibility of SubDomain by putting it to work confining a variety of software components common to Internet servers, both large and small. SubDomain can confine binary-only programs, so long as there is no need for sub-process confinement. If sub-process confinement is required, then the program source needs to be edited and re-compiled to insert appropriate calls to change_hat().

Like the "synthetic user" approach mentioned earlier, SubDomain confinement requires administrator intervention. However, SubDomain confinement is easier for the administrator in the following ways:

Ease of Application: A SubDomain profile does not interfere with any other aspects of the system except the SubDomain mechanism. Thus it is easy to install a SubDomain profile along with the confined program. In particular, because the SubDomain profile is independent of the system the program is being installed on, the profile can be *included* with the program being distributed. In contrast, it is difficult to include a synthetic user in conventional program packages (e.g., tar balls or RPM packages).[2]

- It is easy for the administrator to inspect a SubDomain specification to determine the precise aspects of the system that are exposed to that program. In contrast, the exposure entailed by a synthetic user is non-obvious: the administrator must consider all files that are accessible to "anybody," which is a non-trivial exercise on non-trivial file systems.

The Kernel Wrapper approach [23] provides for confinement scripts that are full Turing-equivalent programs. While this provides extensive flexibility, it also means that the completeness and safety of an inserted kernel wrapper is not amenable to automatic analysis. In contrast, SubDomain profiles are easy to inspect to determine the security implications of updating a SubDomain profile. Furthermore, it is *strictly* safe to install a SubDomain profile where none existed before, because SubDomain strictly limits program privileges.

These factors have important implications for software distribution. Because SubDomain confinement profiles are system independent and guaranteed to be safe to install, it becomes feasible to package SubDomain confinement *with the program itself*. Thus an end user can consider installing a new program on a server appliance, and because of the SubDomain confinement information packaged with the program, the user can understand the security implications of installing that program. In future work, we plan to

[2] Note that bundling synthetic user IDs is exactly the approach taken by qmail [11], which results in excellent security for qmail, but also imposes substantial packaging difficulties that have hampered qmail's spread.

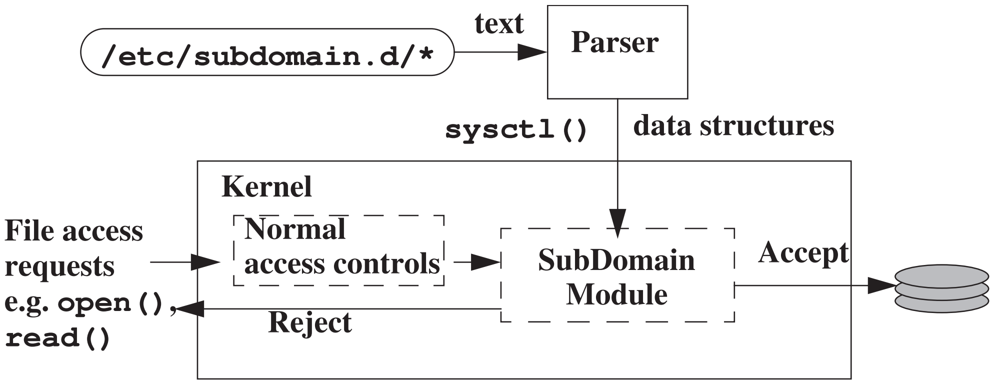

Figure 4: SubDomain implementation

develop future tools that will assist the administrator in determining the security implications of a set of SubDomain confinements

Which programs need to be confined with SubDomain depends on the convenience and security needs of the host system, and thus is an adjustable policy. The administrator can specify which of the following classes of programs must be confined with SubDomain before they are allowed to execute at all:

All Programs: All programs that execute on the host must be associated with a SubDomain, either explicitly, or inherited from a SubDomained parent program. This mode is suitable for bastion hosts.

All Listed User-IDs: All programs running under one of the user-IDs specified by the administrator must be associated with a SubDomain. For instance, the httpd user-ID runs many programs on behalf of the web server, and SubDomain confinement ensures that these programs will not affect other parts of the system. This mode is suitable for confining a potentially vulnerable collection of services on a system that also hosts critical data.

All root Programs: All programs running with a real or effective user-ID of ''root.'' This mode allows a SubDomain profile to be used to achieve the classic goal of breaking up root's all-too-powerful privileges. The (defunct) POSIX 1.e ''capabilities'' model subdivided root's powers into a static set of 32 separate groups of ''capabilities'', and individual programs could assume part of root's powers by flipping on one or more of these sets of capabilities. SubDomain allows arbitrary sets of privileges to be grouped together, rather than accepting the groupings specified by POSIX 1.e.

Only Specified Programs: Only the programs that have a SubDomain specified are thus confined, i.e., ''default allow.'' This mode assumes that all programs on the host are adequately secured *except* for the programs being SubDomained. While not especially secure, this mode is convenient, e.g., for use on a client workstation to run a suspect program recently downloaded from the Internet.

The procedure for confining a program is to start with a null subdomain specification, run the application, observe the system log for complaints about attempts to access files outside the subdomain, and then add those files to the subdomain specification. This procedure is presently manual, because due consideration is required for two stages in this procedure:

Running the application: The application needs to be run under all of the ''kinds'' of input that it is expected to experience in a production environment, i.e., a comprehensive test suite. Determining these inputs requires some knowledge of the application to ensure complete coverage. Failure to provide complete coverage results in a subdomain that is too ''tight'', and the application will occasionally fail to access resources that it needs.

Granting the privilege: We are confining the application precisely because we do *not* trust it, and therefore we cannot automatically assume that every file the application tries to access under test is a legitimate file for the program to access. The file should be included in the subdomain only after due consideration of the security implications.

For applications where source code is available, predicting the set of required resources should be feasible. If anticipating the set of files an application needs to access is truly difficult, then it is quite likely that the application represents a significant security threat, and should not be installed on hosts requiring security.

For applications where source code is not available, a run-time testing methodology must be used to experimentally identify all of the file resources that a program may try to access. To facilitate this, we use the dep program that we developed for the InDependence project [16] (funded by a student grant from USENIX). This program uses strace() to monitor the execution of a subject program, and amasses a list of all the files accessed. dep's use of strace() imposes heavier performance and compatibility overhead than SubDomain, but is none the less sufficient for exploring the file system domain of many programs. To further ease use, dep accumulates files accessed across multiple runs, so that a large test suite can be applied, and then the list of files accessed inspected once at the end of testing.

```
/home/httpd/cgi-bin/Count.cgi   {
  /etc/ld.so.cache                         r ,
  /lib/lib*                                r ,
  /lib/ld-linux.so.2                       r ,
  /etc/nsswitch.conf                       r ,
  /etc/wwwcounter.conf                     r ,
  /etc/localtime                           r ,
  /var/log/httpd/wwwcount/wwwcount_log     rw ,
  /var/lib/wwwcount/*                      r ,
  /var/lib/wwwcount/data/*                 rw ,
}
```

Figure 5: SubDomain for wwwcount CGI script

An example subdomain profile is shown in Figure 5, providing all of the resources needed to run the wwwcount CGI program (a popular web page hit counter program). Note the use of simple globbing to reduce the size of the subdomain specification when access to an entire directory is required. Figure 6 shows a more elaborate profile for the Apache web server itself, under a particular configuration. A list of some of the programs that we have confined and tested, along with the size of their subdomains, are listed in Table 1.

```
/usr/local/apache/bin/httpd {
  /                                              r  ,
  /dev/null                                      rw ,
  /dev/urandom                                   r  ,
  /etc/group                                     r  ,
  /etc/hosts                                     r  ,
  /etc/host.conf                                 r  ,
  /etc/ld.so.cache                               r  ,
  /etc/localtime                                 r  ,
  /etc/nsswitch.conf                             r  ,
  /etc/passwd                                    r  ,
  /etc/resolv.conf                               r  ,
  /home/httpd/perl/*                             r  ,
  /lib/*                                         r  ,
  /usr                                           r  ,
  /usr/lib/gconv/ISO8859-1.so                    r  ,
  /usr/lib/gconv/gconv-modules                   r  ,
  /usr/lib/perl5/5.00503/*                       r  ,
  /usr/lib/perl5/site_perl/5.005/i386-linux/*    r  ,
  /usr/local                                     r  ,
  /usr/local/apache                              r  ,
  /usr/local/apache/conf/*                       r  ,
  /usr/local/apache/htdocs/*                     r  ,
  /usr/local/apache/logs*                        wl ,
  /usr/share/locale/en_US/*                      r  ,
  /usr/share/locale/locale.alias                 r  ,
}
```

Figure 6: SubDomain for Apache Web Server.

SubDomain Performance

Here we present a variety of SubDomain performance measurements. The next section describes our microbenchmarks on mediated system calls, and the section after that describes our macrobenchmarks on a confined PERL script interpreted by the mod_perl Apache module.

Microbenchmarks

Here we use the usual benchmarking technique to measure affected system calls by crafting programs that issue each system call 10,000 times, run the programs several times, discard the first run to avoid cold cache effects, and average the remainder. All tests were performed on a dual-processor Pentium III 700 MHz, with 256 MB of RAM. Table 2 summarizes these results. We include measurement of the get_pid() system call as a baseline for comparison against the change_hat() system call, as get_pid() is commonly regarded as the simplest system call.

As expected, he major overhead appears in the open(), exec() and change_hat() system calls, where SubDomain is checking the action against the subdomain specification for the confined process.

Program	Size of SubDomain
Simple bash shell script	31 files
PHF CGI program	14 files
CGI Mail program	7 files
htsearch CGI program	11 files
wwwcount CGI program	10 files
Apache web server	33 files
lpd	16 files
lpq	10 files
lpc	11 files
Postfix Mail Delivery Agent	15 files
Postfix-script helper program	65 files

Table 1: SubDomain-confined programs.

Macrobenchmarks

Our macrobenchmark is SubDomain confinement of a PERL script to be executed via the mod_perl Apache module, thus exercising SubDomain's capability to confine *active content* scripts. To exercise the web server's cache, we replicated the PERL script 1000 times, and used the Webstone performance benchmark to measure the overhead cost of PERL scripted web pages protected with SubDomain vs. without protection. The PERL script itself reads two files with some busy-work in between, simulating a script that fetches a "container" template from one file, HTML content from another file, and does some interim processing to merge the two, e.g., compute a

System Call	Standard Cost	SubDomain Cost	% Overhead
fork()	295	295	0%
exec()	1387	1487	7%
open()	3.71	5.39	45%
get_pid() vs. change_hat()	1.81	4.70	159%

Table 2: SubDomain Microbenchmarks in microseconds.

```
/perl/0/cgitest-001.cgi {
 /usr/lib/perl5/site_perl/5.005/i386-linux/Apache/Registry.pm    r
 /etc/localtime                                                  r
 /usr/lib/perl5/5.00503/*                                        r
 /home/httpd/perl/0/cgitest-001.cgi                              r
 /home/httpd/perl/0/cgitemplate-001.html                         r
 /home/httpd/perl/0/cgidata-001                                  r
 /var/log/httpd/*                                                w
}
```

Figure 7: Test PERL script's SubDomain profile.

hit counter. The SubDomain profile for this script is shown in Figure 7.

The test environment used the same dual-processor Pentium III 700 MHz server with 256 MB of RAM, and a private network (crossover cable) via 100 Mbit ethernet.

The test results are shown in Table 3, measured for 5 to 10 concurrent client connections. Tests were run twice, and the results averaged. For all cases, the SubDomain overhead is between 1% and 2%, i.e., in the noise range.

Related Work

Here we describe work that, similar to SubDomain, specifically attacks the problem of confining suspect programs. Despite the age of the notion of least privilege [33], much of this work has emerged relatively recently. It is our conjecture that this is a result of a shift in emphasis from defending secrecy (the dominant concern for military organizations) to defending *integrity* (the dominant concern for Internet-connected businesses) and the emergence of the notion of *survivability* [35]. This list of related work is necessarily partial, as the total body of related work is very large.

TRON

The TRON system [10] is a kernel enhancement for ULTRIX that can confine a program's execution to a protection domain consisting of a finite set of *capabilities* in the form of file names. TRON adds the tron_fork() system call, which functions exactly like the classic fork() system call, except that it specifies the protection domain as an extra argument. TRON is semantically most similar to SubDomain: the protection domains are the same (sets of files) and are similarly applied to host programs, orthogonal to user privileges. The major differences are:

- TRON is discretionary, while SubDomain is mandatory. TRON provides user commands to run programs in a confined domain, while SubDomain always runs a specified program in a confined domain. Thus in the usual DAC vs. MAC trade-off, TRON is more convenient for individual users, while SubDomain is more convenient for securing entire systems, e.g., server appliances.
- TRON's finest granularity is the ULTRIX process; it cannot confine loadable modules.

Janus

Janus [25] is a user-level mechanism for confining programs to a specific set of resources. Intended to confine "helper" applications run from within a Web browser, Janus uses the ptrace() system call and a monitoring process to mediate all system calls made by the helper application. If the action proposed by the helper application violates a policy set by the user, then the monitoring process rejects the request. This

Test	# of Clients	Connection Rate	Avg. Response Time (ms)	Avg. Client Throughput
Standard	5	75.97	66.5	26.29
SubDomain	5	75.19	66.5	26.02
% Overhead		1%	0%	1%
Standard	6	78.14	77	27.04
SubDomain	6	76.56	78	26.49
% Overhead		2%	1.3%	2%
Standard	7	78.38	89	27.13
SubDomain	7	77.24	90.5	26.73
% Overhead		1.45%	1.7%	1.5%
Standard	8	78.26	102	27.08
SubDomain	8	76.71	104	26.54
% Overhead		2%	2%	2%
Standard	9	78.24	115	27.08
SubDomain	9	77.02	116.5	26.66
% Overhead		1.6%	1.3%	1.6%
Standard	10	78.43	127	27.15
SubDomain	10	77.07	129.5	26.67
% Overhead		1.7%	2%	1.7%

Table 3: SubDomain macrobenchmarks with WebStone.

approach requires four system calls to be executed to effect one confined system call.

Java 2 Security

The Java 2 security model [4] allows the JVM to be configured to assign particular capabilities to designated Java *classes*, similar to the SubDomain notion of assigning file system capabilities to programs. This is an enrichment over the original Java security model [26] which assigned one fixed set of capabilities to remotely-loaded applets (almost nothing), and another fixed set of capabilities to locally-loaded applets (almost everything).

The Java 2 security mechanism is notable as the only system other than SubDomain capable of confining sub-process components, in that Java classes are typically smaller than the host OS processes. Naturally, the Java 2 security model does not apply to non-Java native executables.

chroot Jail

The chroot() system call[3] makes the argument directory be the effective root directory, i.e., ''/'' for the invoking process. The point of this operation is that the file system domain for the affected process is now limited to the contents of the argument directory. Any files that the application needs to access must be placed inside the chroot directory, or the access will fail.

The chroot technique is a popular form of confinement, in large part because standard kernels (e.g., Linux) support it. However, chroot has defects in all three of the dimensions a security enhancement should address:

Security: chroot jails are resistant to oblivious attempts to escape the jail, i.e., attempts to access files that are not accessible within the jail. However, if the attacker can execute their own code within the chroot jail, it is fairly easy to break the jail and access outside files. Thus jailed programs generally cannot be trusted with strong privileges, i.e., it is insecure to depend on chroot to confine a root process.

Compatibility: Each chroot'd program must have the necessary components of the file system replicated within its jail, which is problematic if the program requires access to a large, complex set of files, i.e., shell scripts need all invoked programs replicated into the chroot jail. Thus setting up a chroot jail can be a lot of tedious, complex work. The chroot technique also breaks programs that need to interact with other parts of the system.

Performance: Because chroot jails require duplication of all resources needed by the jailed program (soft or hard links could be used as escape routes) they consume excessive disk space and file system buffer cache space.

[3]''man chroot'' on most UNIX Systems

Type Enforcement

The type enforcement work [13, 7, 8] has recently been extended to provide better support for program confinement. *Kernel hypervisors* [28] provide a facility for installing small state machines that intercept kernel system calls and enforce a security policy. Such a facility can be viewed as a tool that could be used to build a SubDomain-like least privilege system. Fraiser, Badger and Feldman provide a similar tool for building security policy enforcement automata [23]. SubDomain provides the following key advantages over this technique:

Parsimony: SubDomain is much simpler than the TE and DTE implementations; the SubDomain kernel code is approximately 1/10 the size of the DTE kernel patch. Simplicity is critical in security systems.

Safety: The DTE Wrapper system [23] allows code to be inserted into the operating system to perform mediation. While this is a powerful technique, it is also dangerous: *malicious* DTE wrapper code could just as easily be inserted. In contrast, SubDomain profiles are easy to inspect to determine the security implications of updating a SubDomain profile. Furthermore, it is *strictly* safe to install a SubDomain profile where none existed before, because SubDomain strictly limits program privileges.[4]

Application-Specific Mechanisms

Various application environments provide their own least privilege-like mechanisms. For instance, the PERL interpreter includes a facility known as ''taint'', in which input provided to the PERL script *cannot* be used to formulate an action (i.e., system() operation) unless it has been ''adequately'' inspected by the PERL script [41]. PERL also includes a ''safe PERL'' facility, where in the programmer can specify a set of PERL operators that the script may not use.

Another application-specific least privilege mechanism is the notion of ''wrappers.'' For example, CGI Wrappers [31] causes a CGI script to be run with the user-ID of the script owner, rather than the user-ID of the web server. Combined with the synthetic user-ID notion described previously, CGI Wrappers can construct a least privilege environment for CGI scripts.

PACLs: Program-based Access Control Lists

We believe PACLs [42] to be the first instance of an access control system based on the program performing the operation. The PACL system is the exact dual of the SubDomain notion: files have an access control list that enumerates programs that are permitted to operate on that file. A simulated PACL system was built and evaluated, but an actual PACL system was never finished.

[4]This observation due to Blaine Burnham.

Status & Availability

The implementation is not complete with respect to the description in this paper.

- The absolute and relative sub-domains described earlier are not complete: child processes either inherit the parent's profile, or use their own profile if one is specified.
- The multiple modes of requiring SubDomain confinement described previously is only partially implemented. The implementation currently supports ''paranoid'' mode where all processes must have SubDomain confinement, and ''open'' mode, where only the programs that are specified are confined by SubDomain.

SubDomain is implemented for Linux, and is available from http://immunix.org . The kernel enhancement portion is licensed under the GPL, and the non-kernel portions are proprietary to WireX but available for free for non-commercial use.

Conclusions

Vulnerable software is a major security problem, mandating constant system administrator attention to keep systems up to date with vendor-supplied security patches. This is especially problematic for complex Internet servers, which are required to provide extensive services to anyone on the Internet. Some form of confinement mechanism to approximate least privilege is the generic solution, but often imposes more costs than administrators deploying in ''internet time'' can bear. Our SubDomain confinement mechanism advances over previous confinement work, simplifying both implementation and administration overheads by confining *programs* instead of *users*.

This approach enables SubDomain confinement to be packaged with programs, in contrast with confinement mechanisms that are bound to the system. SubDomain also provides fine-grained protection, confining software components *finer* than the host OS process, providing the unique capability to protect potentially vulnerable server *modules* such as Microsoft's Front Page Extensions to the Apache web server. We have implemented and tested the system, showing that it provides all three essential properties of a security enhancement: enhanced security, software compatibility, and preserved performance.

Author Information

Dr. Crispin Cowan is co-founder and Chief Research Scientist of WireX Communications, Inc., and is a Research Assistant Professor at the Oregon Graduate Institute, where he teaches graduate courses in system security. His research focuses on making existing systems more secure without breaking compatibility or compromising performance. Professor Cowan has authored 28 refereed publications, including those describing the StackGuard compiler for defending against buffer overflow attacks. Reach him electronically at crispin@wirex.com .

Steve Beattie is one of the original developers and the current maintainer of the StackGuard compiler enhancement. He is employed in the Advanced R&D Group at WireX Communications, Inc., who graciously pays him to work on such fun projects as StackGuard and SubDomain. He received a Masters Degree in Computer Science from the Oregon Graduate Institute, and was previously employed as a Jack-of-all-Trades Sysadmin in a Large Dead-Tree Publishing Corporation.

Calton Pu received his Ph.D .from University of Washington in 1986 and served on the faculty of Columbia University and Oregon Graduate Institute. Currently, he is holding the position of Professor and John P. Imlay, Jr. Chair in Software at the College of Computing, Georgia Institute of Technology. He is leading the Infosphere project that combines his research interests. First, he has been working on next-generation operating system kernels to achieve high performance, adaptiveness, security, and modularity, using program specialization, software feedback, and domain-specific languages. This area has included projects such as Synthetix, Immunix, Microlanguages, and Microfeedback, applied to distributed multimedia and system survivability. Second, he has been working on new data and transaction management by extending database technology. This area has included projects such as Epsilon Serializability, Reflective Transaction Framework, and Continual Queries over the Internet. He has published more than 30 journal papers and book chapters, 100 conference and refereed workshop papers, and served on more than 40 program committees. He is currently an associate editor of IEEE TKDE, DAPD, and IJODL.

Greg Kroah-Hartman is one of the main Linux USB developers, and the current Linux USB Serial and USB Bluetooth driver maintainer. He is also the author and maintainer of the Linux usbview program which is being shipped in most of the major Linux distributions. His free software is being used by more people than any closed source projects he has ever been paid to develop. He is currently employed in the Advanced R&D group at WireX Communications Inc, and has a Bachelors Degree in Computer Science.

Perry Wagle received his MS in Computer Science at Indiana University in 1995, and in 1997 he headed to the Oregon Graduate Institute to join the Immunix project's survivability research, where among other things, he was was the primary programmer of the StackGuard enhancement to GCC. Rather than go with WireX like the rest of Immunix last year, he stayed at OGI to work on the Infosphere project. He is interested in applying programming language technology to operating systems problems. For example, survivability presents an interesting problem: how effectively can you transform or assist legacy code that is intolerate to security faults so that it responds sensibly to attacks?

Virgil D. Gligor received his B.Sc., M.Sc., and Ph.D. degrees from the University of California at Berkeley. He has been at the University of Maryland since 1976, and is currently a Professor of Electrical and Computer Engineering. He has worked in the areas of acces control on several UNIX systems an is the co-designer of two automated tools, one for covert-channel analysis of (C language) source code (written in PROLOG), and the other for penetration analysis, which have been used by IBM Corporation for analysis of Secure Xenix and for TIS Inc Trusted Xenix. His work helped define precisely the notion of the denial of service, and received the Best Research Paper Award at the 1988 IEEE Symposium on Research in Security and Privacy for research work in this area (paper co-authored with his graduate student C.F. Yu). He is the co-author of several US patents in the areas of intrusion detection, penetration analysis methods and tools, and role-based access control.

References

[1] Alpern, Bowen and Fred B. Schneider, ''Defining Liveness,'' *Information Processing Letters*, 21(4):181-185, 1985.

[2] Amoroso, Edward, *Fundamentals of Computer Security Technology,* Prentice Hall, Englewood Cliffs, NJ, 1994.

[3] Anonymous, *The Java Web Server Architecture Overview*, http://www.javasoft.com/products/java-server/documentation/webserver1.1/ , 1997.

[4] Anonymous, *JDK 1.2 Security*, http://java.sun.com/products/jdk/1.2/docs/guide/security/index.html , March 1998.

[5] Assorted, *NCSA HTTPd Tutorial: Server Side Includes*, http://hoohoo.ncsa.uiuc.edu/docs/tutorials/includes.html .

[6] Assorted, *PHP Hypertext Processor,* http://php3.org/ .

[7] L. Badger, D. F. Sterne, et al., ''Practical Domain and Type Enforcement for UNIX,'' *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1995.

[8] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat, ''A Domain and Type Enforcement UNIX Prototype,'' *Proceedings of the USENIX Security Conference*, 1995.

[9] Brian Behlendorf, Roy T. Fielding, Rob Hartill, David Robinson, Cliff Skolnick, Randy Terbush, Robert S. Thau, and Andrew Wilson, *Apache HTTP Server Project*, http://www.apache.org .

[10] Andrew Berman, Virgil Bourassa, and Erik Selberg, ''TRON: Process-Specific File Protection for the UNIX Operating System,'' *Proceedings of the 1995 Winter USENIX Conference,* USENIX Association, 1995.

[11] D. J. Bernstein, *qmail*, http://cr.yp.to/qmail.html , 1990.

[12] M. Bishop and M. Digler, ''Checking for Race Conditions in File Accesses,'' *Computing Systems*, 9(2):131-152, http://olympus.cs.ucdavis.edu/bishop/scriv/index.html , Spring 1996.

[13] W. E. Bobert and R. Y. Kain, ''A Practical Alternative to Hierarchical Integrity Policies,'' *Proceedings of the 8th National Computer Security Conference*, Gaithersburg, MD, 1985.

[14] CERT, *Advisory CA-96.06: Vulnerability in NCSA/Apache CGI Example Code,* ftp://info.cert.org/pub/cert_advisories/CA-96.06.cgi_example_code , September 1996.

[15] CERT, *Advisory CA-98.05: Multiple Vulnerabilities in BIND,* ftp://info.cert.org/pub/cert_advisories/CA-98.05.bind_problems , May 1998.

[16] Crispin Cowan, Ryan Finnin Day, and Hao Zhao, *InDependence: Automating the Discovery of Application Dependencies,* http://www.cse.ogi.edu/DISC/projects/independence, 1997.

[17] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang, ''StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks,'' *7th USENIX Security Conference*, pages 63-77, San Antonio, TX, January 1998.

[18] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole, ''Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade,'' *DARPA Information Survivability Conference and Expo (DISCEX)*, January 2000; Also presented as an invited talk at SANS 2000, Orlando, FL, http://schafercorp-ballston.com/discex , March 23-26, 2000.

[19] Michele Crabb, ''Curmudgeon's Executive Summary,'' *The SANS Network Security Digest,* Michele Crabb, editor, Contributing Editors: Matt Bishop, Gene Spafford, Steve Bellovin, Gene Schultz, Rob Kolstad, Marcus Ranum, Dorothy Denning, Dan Geer, Peter Neumann, Peter Galvin, David Harley, Jean Chouanard, SANS, 1997.

[20] Drew Dean, Edward W. Felten, and Dan S. Wallach, ''Java Security: From HotJava to Netscape and Beyond,'' *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, http://www.cs.princeton.edu/sip/pub/secure96.html , 1996.

[21] eEye, *IIS Remote Hole,* http://www.eye.com/database/advisories/ad06081999/ad06081999.html , June 1999.

[22] David F. Ferraiolo and Richard Kuhn, ''Role-Based Access Control,'' *Proceedings of the 15th National Computer Security Conference, Baltimore*, MD, October 1992.

[23] Tim Fraser, Lee Badger, and Mark Feldman, ''Hardening COTS Software with Generic Software Wrappers,'' *Proceedings of the IEEE*

Symposium on Security and Privacy, Oakland, CA, May 1999.
[24] Neal Glew and Greg Morrisett, "Type-Safe Linking and Modular Assembly Language," *Twenty-Sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 250-261, San Antonio, TX, http://www.cs.cornell.edu/talc/papers.html , January 1999.
[25] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer, "A Secure Environment for Untrusted Helper Applications," *6th USENIX Security Conference*, San Jose, CA, July 1996.
[26] James Gosling and Henry McGilton, "The Java Language Environment: A White Paper," http://www.javasoft.com/docs/white/langenv/, May 1996.
[27] J. A. McLean, "A General Theory of the Composition for Trace Sets Closed Under Selective Interleaving Functions," *Proceedings of the IEEE Symposium on Security and Privacy*, pages 79-93, Oakland, CA, May 1994.
[28] Terrance Mitchem, Raymond Lu, and Richard O'Brien, "Using Kernel Hypervisors to Secure Applications," *Proceedings of the Annual Computer Security Application Conference*, December 1997.
[29] Mudge, *How to Write Buffer Overflows,* http://l0pht.com/advisories/bufero.html, 1997.
[30] George C. Necula and Peter Lee, "Safe Kernel Extensions Without Run-Time Checking," *Proceedings of the USENIX 2nd Symposium on OS Design and Implementation (OSDI'96)*, http://www.usenix.org/publications/library/proceedings/osdi96/necula.html , 1996.
[31] Nathan Neulinger, *CGIWrap: User CGI Access*, http://www.unixtools.org/cgiwrap/ , 1997.
[32] Aleph One, "Smashing The Stack For Fun And Profit," *Phrack*, 7(49), November 1996.
[33] Jerome H. Saltzer and Michael D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, 63(9), November 1975.
[34] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role Based Access Control Models," *IEEE Computer*, pages 38-47, February 1996.
[35] Howie Shrobe, *ARPATech '96 Information Survivability Briefing,* http://www.darpa.mil/ito/ARPATech96_Briefs/survivability/survive_brief.html , May 1996.
[36] Marc Slemko, *Microsoft FrontPage 98 Security Hell,* http://www.worldgate.com/marcs/fp/ , October 1997.
[37] Robert E. Strom and Shaula Alexander Yemini, "Typestate: A Programming Language Concept for Enhancing Software Reliability," *IEEE Transactions on Software Engineering*, 12(1):157-171, January 1986.
[38] Linus Torvalds, et al., *Linux Operating System,* http://www.linux.org/ .
[39] United States Department of Defense, *Reference Manual for the Ada Programming Language ANSI/MIL-STD-1815A-1983*, United States Department of Defense, February 1983.
[40] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham, "Efficient Software-Based Fault Isolation," *Proceedings of the Fourteenth ACM Symposium on Operating System Principles (SOSP'93)*, pages 203-216, Asheville, NC, December 1993.
[41] Larry Wall, Tom Christiansen, and Randal L. Schwartz, *Programming Perl,* O'Reilly & Associates, Inc., 2nd edition, 1996.
[42] D. R. Wichers, D. M. Cook, R. A. Olsson, J. Crossley, P. Kerchen, K. Levitt, and R. Lo, "PACL's: An Access Control List Approach to Anti-viral Security," *Proceedings of the 13th National Computer Security Conference*, pages 340-349, Washington, DC, October 1-4 1990.