Using Operating System Wrappers to Increase the Resiliency of Commercial Firewalls

Jeremy Epstein
webMethods, Inc.
jepstein@webMethods.com

Linda Thomas
webMethods, Inc.
lthomas@webMethods.com

Eric Monteith
NAI Labs
eric_monteith@nai.com

Abstract[1]

Operating system wrappers technology provides a means for providing fine grained controls on the operation of applications software. Application proxy firewalls can gain from this technology by wrapping the proxies, thus preventing bugs (or malicious software) in the proxy from subverting the intent of the firewall. This paper describes several experiments we performed with wrappers and firewalls, using several different firewalls and types of wrappers.

1 Introduction

Access controls in operating systems are usually at a coarse level and frequently do not cover all types of resources in the system. For example, UNIX systems control access to files, but the only controls on sockets limit non-root processes from binding low numbered sockets. Operating system wrapper technologies (henceforth "wrappers"), including those described in [Jones], [Fraser], [Balzer], among others, allow specifying the behavior of application processes to an arbitrary level of granularity.[2]

While wrapper technology is aimed at constraining the behavior of applications on end systems (especially clients, and possibly also servers), it is also applicable to security devices such as firewalls. As part of the DARPA Information Assurance program, we have performed a series of experiments using different types of wrappers to constrain the behavior of several different firewall products. This paper describes the results of those experiments, and points to directions for future research.

The remainder of this paper is organized as follows. Section 2 describes our motivation for developing firewall wrappers. While this paper assumes a basic understanding of wrapper technology, Section 3 provides a synopsis of what wrappers are and how they work, and describes some of the differences between the wrappers technology developed by NAI Labs [Fraser] and the wrappers technology developed by the Information Sciences Institute (ISI) [Balzer]. Section 4 describes how we wrapped the Gauntlet Internet Firewall (for which we had design information and source code available) using the NAI Labs wrappers. Section 5 describes our experiences in using the NAI Labs wrappers to wrap firewalls for which we had no source code or design information. Section 6 describes how we wrapped the Gauntlet Internet Firewall using the ISI wrappers, and as such is a parallel to Section 4. Section 7 gives our current status and availability of our prototypes. Section 8 concludes the paper.

2 Motivation

Application level firewall proxies are fragile, and are growing ever more complex. Customers demand increasing functionality, including the ability to perform tasks such as virus scanning, limits on addresses visited (e.g., to prevent access to pornographic web sites), and detailed scanning of protocols to prevent outsiders from exploiting vulnerabilities in host systems. As the proxies become increasingly complex, the likelihood of flaws that allow security breaches increases. For example, it is likely that there are opportunities in most firewall proxies for buffer overrun attacks.

As the number of protocols increases, proxies are increasingly written by people without sufficient training in writing safe software. End users want to write their own proxies, since they can do it more rapidly than waiting for a firewall vendor to include a suitable proxy in the product. While both vendors and end users make reasonable efforts to ensure that proxies are not being written by hostile developers (who might insert backdoors or other malicious software), it is likely that such capabilities have been inserted in at least some proxies. Finally, when this effort was begun, there was significant concern about the then-impending Y2K crisis, and as such

[1] The work described in this paper was performed while all three authors were associated by NAI Labs.

[2] The term "wrappers' is overloaded in the security field. In this paper, it means functions that intercept system calls and perform mediation. This is different from TCP Wrappers [Venema] which are a program between *inetd* and the service provider daemons, but do not attempt to intercept system calls.

there was concern in DoD and elsewhere that backdoors were being inserted as a byproduct of Y2K remediation, including in security products such as firewalls.

Since a single faulty proxy can endanger an entire firewall (and the network behind it), it is important to constrain the damage done by an errant proxy. The historical approach to such threats would be to use good software engineering techniques (including code inspection), personnel security (such as clearances), and improved testing. However, these are not realistic in today's "Internet time" commercial products environment. Additionally, we want to constrain proxies for which we may not have source available, since that allows us to use a variety of different products.[3]

Our goal, then, was to provide a method for constraining proxies without requiring source code. The constraints should be simple enough (i.e., much simpler than the proxy itself), so that the wrapper can reasonably be subjected to a detailed correctness analysis. The wrappers technology allows us accomplish these goals.

Some argue that it is more effective to harden the operating system used in the firewall than to add a layer of protection such as wrappers. In a certain sense, wrappers *are* hardening the operating system: they allow for a more granular control of capabilities, but in a general purpose way that can serve not only firewalls but also other types of computer systems.

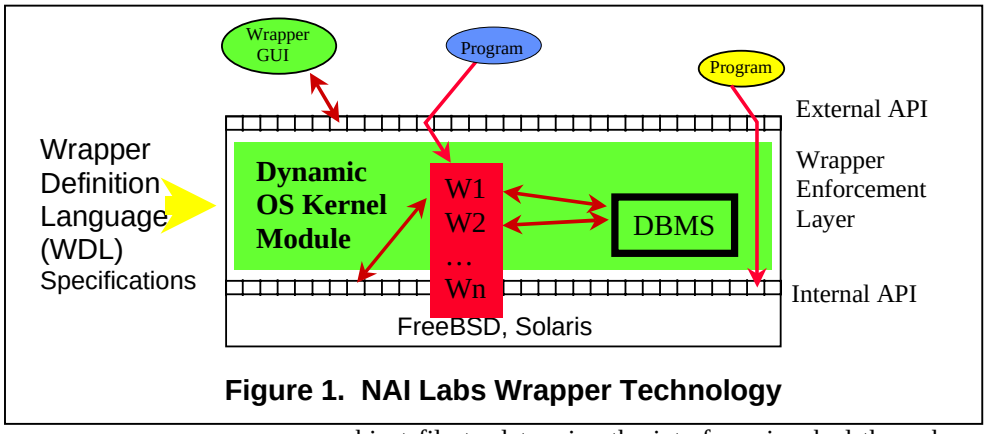

Figure 1. NAI Labs Wrapper Technology

3 Wrappers Synopsis

The idea of wrappers technologies is to provide for relatively small specifications of the allowed behavior of software. The NAI Labs wrapper technology is currently implemented for UNIX and NT; the ISI wrappers technology is currently available for NT only.

The premise of all wrappers technology is that the application being wrapped should be unaware of the wrapping, and should not need any modifications to be wrapped. Applications may become indirectly aware of wrappers because operations that succeed on an unwrapped system may fail on a wrapped system, but this is an unavoidable side effect.

In some cases, it is possible to determine *a priori* what system calls will be used (e.g., by statically examining the object file to determine the interfaces invoked through a DLL or a shared library). In the more general case, though, some system call interfaces may have been statically linked, in which case such external references will not appear, or system calls may be called directly without going through APIs. Thus, it is important to catch the system calls where they enter the operating system, and not at an API level.

The UNIX version of the NAI Labs wrapper technology uses a kernel loadable module to intercept all system calls as they are made, and pass them to a wrapper based on criteria specified when the wrapper is loaded. We will refer to the former as the Wrapper Enforcement Layer (WEL) and the latter as the Wrapper Specification (WS). A system can have many different wrapper specifications loaded, with different activation criteria. For example, a WS might apply to all programs run by a specific user, or to all instances of programs run from a particular directory. Figure 1 shows the operation of the wrappers technology. Because the interception occurs inside the kernel, the NAI Labs wrappers cannot be bypassed by malicious code.

The NT version of the NAI Labs wrapper technology and the ISI wrappers technology rely on intercepting system calls in user space, before they are passed to the NT kernel. Both technologies strive to make it as difficult as possible to bypass the wrapper, but since they operate in user space, they can be bypassed by a sufficiently determined adversary.

Wrapping is not a panacea. The following sections describe some of the limitations of wrapping.

3.1 Limitations on Correct Behavior

Wrappers cannot stop all malfunctioning software. For example, if a WS were developed for a web server, the WS could reasonably prevent the web server from retrieving files that it did not have any need to access, but it could not prevent the web server from retrieving a configuration file and transmitting it to a remote site, which would be an undesirable action, but one not visible

[3] While some object to the very concept of running proxies without source code available, it is the essence of today's commercial firewall market. Therefore, it is important to address, even if it is not the ideal situation.

at the system call interface level. Similarly, if a wrapped program should allow certain users access to a particular file, but should deny access to other users, it is effectively impossible to enforce that constraint in the WS.

The premise of wrappers is that the WS should be much simpler than the software it wraps. If the WS needs to know as much and keep as much state as the application being wrapped, then the WS will be no simpler than the application. In this case, wrappers degenerate to a variation on N-version programming, which is both expensive and not particularly resilient against flaws.

3.2 Flaws in the Underlying Operating System

In the UNIX version of NAI Labs wrappers, the WEL and the WS run as part of the operating system. As a result, flaws in the wrapper can bring down the operating system. Additionally, flaws in the operating system can subvert the wrappers. For example, if there is a flaw that allows an application program to subvert or bypass the wrapper, the wrappers technology will be unable to stop it.

3.3 Limitations of Knowledge

Both the UNIX and NT versions of NAI Labs wrappers use a Wrapper Definition Language (WDL), which provides "characterization" of system calls (i.e., grouping related calls together). The characterization also names and types parameters. This characterization allows the WS developer to be ignorant of the system call details, and instead to focus on the classes of operations. However, if the characterization is incorrect (i.e., it misses one or more related system calls), then not all calls will be captured.

By contrast, the ISI wrappers require the developer to write a DLL that specifies each interface with its parameters. If the developer omits an interface (or makes an error in specifying parameters), the WS will not have the expected results. ISI wrappers do not provide a characterization capability, thus requiring the developer to enumerate each interface to be intercepted.

Both NAI Labs wrappers and ISI wrappers have capabilities for composition, although we did not use that facility in this effort.

In either case, though, the skill of the WS developer is critical in the quality of the resulting WS. For this reason, WSs for security critical systems like firewall proxies need to be vetted particularly carefully.

3.4 Total Wrapping

Because wrappers are just programs on a computer system, they are only as good as their protection. That is, even if a wrapper cannot be bypassed, if an attacker can modify or delete the WEL or WSs, then they can subvert the wrapper indirectly. For this reason, one of our efforts is to develop a "total wrapper" for Gauntlet. A total wrapper consists of a WS for the key parts of the system (e.g., for the proxies) plus a separate WS for everything else on the system that prevents interference with the WSs, the WEL, and the proxies themselves. The result is a system where an attacker who breaks through a proxy may run amok within the system, but will be unable to interfere with the wrappers or the proxies themselves.

For example, the total wrapper (also known as an *interference preventer*) might prevent anyone from binding the ports used by the wrapped proxies, modifying the WEL, the WSs, the proxy software, the configuration files used by the proxy, and shared libraries used by the proxy, etc. It must also constrain those programs with root privilege, limiting them to performing only those operations which will not interfere with the remaining wrappers.[4]

4 Gauntlet Wrappers Using NAI Labs Wrappers

As described above, wrappers provide the means to constrain execution of arbitrary programs, including those running as root. In an application proxy-based firewall such as Gauntlet, proxies are started as root, perform key initialization functions (e.g., binding to low-numbered TCP ports), and then switch to an unprivileged user ID for "normal" operation. However, there are several risks remaining:

- If the proxy contains incorrect code that causes it to perform more than just binding its reserved port before relinquishing root permission, then it could perform arbitrary damage.
- If the proxy contains incorrect code that causes it to perform incorrect operations which are still allowed with its unprivileged ID, it might disclose information which should be restricted.
- If the proxy is vulnerable to buffer overruns, it may be possible to cause it to perform undesired functions (e.g., creating a shell on the firewall machine).

[4] This begs the question of how wrappers are installed in the first place. Initially, any program with root privilege can install a wrapper. When the wrapper is activated for a process, its first action is to disable the capability to install or otherwise modify the wrapper configuration. Thus, even if the process does not give up the root privilege, the wrapper will prevent itself from being subverted. This can be seen in the use of the *wrapperSetupAllowed* variable in Figure 3.

These risks are exacerbated when proxies are written by programmers who are not familiar with the security implications of proxy development. In particular, proxies written by end users (rather than firewall developers) may be more likely to have such flaws.

Proxies should be protected so they operate within a restricted subset of available system calls. The ideal subset should be the set of calls used by a proxy. However, without source code the only way to determine the system calls and arguments used is through a system call wrapper that captures the call profile, thus generating a report of what system calls are used by a particular program. While we had source code to Gauntlet available, we wanted to develop the wrapper as if source code were unavailable (in preparation for the experiment described in Section 5). The completeness of the system call set depends upon the ability of the person exercising the proxy to hit all code paths. In reality, this analysis will usually be incomplete because the conditions needed to exercise a code path will not be known.

Although the firewall's standard policy is to disallow that which is not explicitly allowed, this probably should not be the method used with system calls. Instead, it makes more sense to rely on the wrapper system call classification to, in general, allow or disallow entire classes of operations. Trying to create too precise a wrapper would only result in breaking the proxy, either because of previously unseen code paths or updated versions of the firewall. In cases where it is unclear from initial analysis whether a call or set of operations should be allowed, the wrapper could allow the operation but log the operation so that subsequent analysis can be done to define a more precise wrapper.

For example, well behaved proxies may need to *fork* children (the Gauntlet HTTP proxy does this), but they should not need to *exec* other executables. In general, proxies should not need to write to the file system though specific proxies may need to. Preventing file system *write* operations where possible will prevent subverted proxies from damaging the file system. This includes both normal file writing and directory modifying operations such as *link*, *unlink*, and *rmdir*.

Most proxies need to read firewall related configuration files. They may also need access to certain system-wide configuration files, which are usually in `/etc`. Even if they are not running in a chrooted environment, proxies can be restricted to a well-defined set of files they are allowed to read.

The very essence of a proxy is network I/O so they will need to perform socket related system calls. Depending on the proxy, it may be possible to restrict the ports it is allowed to access so that a proxy cannot poke extra holes in the firewall.

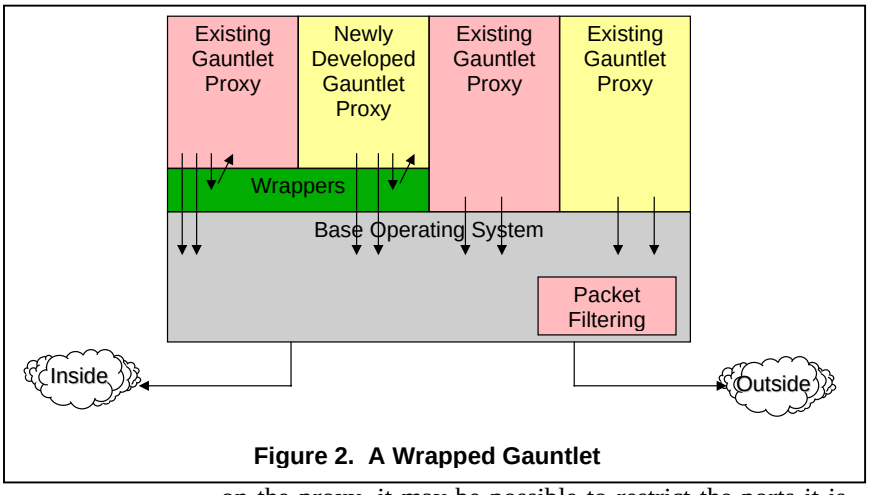

Figure 2. A Wrapped Gauntlet

Proxies do not need most "root" operations even if they are required to run as root. For those proxies that need root-only system calls, they should be restricted to only the needed calls and the arguments should be validated.

Our first step was to create wrappers for some of the more common firewall proxies, such as the HTTP, SMTP, and FTP proxies. In particular, the HTTP proxy is fairly complex, and as such is more likely than others to have security flaws. Figure 2 shows a wrapped Gauntlet with some existing and some new proxies wrapped.

The proxy wrappers are fairly simple. For example, the set of restrictions enforced for the HTTP proxy are:

- Disallow all system administration calls except those specifically exempted below.
- Allow *setuid* only to the value from the Gauntlet configuration (uucp by default).
- Allow *setgrp* only to the value from the Gauntlet configuration (6 by default).
- Disallow writing to the file system.
- Disallow file system operations like *unlink*, *link*, *rmdir*, etc.
- Allow reads only from the Gauntlet installation directory (`/usr/local/etc`).
- Allow *bind* only to socket configured by Gauntlet (80 by default).
- *Fork* is allowed but *exec* is not.

The resulting wrapper to implement this policy is also fairly simple, and is shown in Figure 3 (certain preliminary definitions have been omitted in the interest of space).

It is up the author of the wrapper whether system calls should be allowed or denied by default; in this wrapper, the default is that system calls are allowed. Thus, there is

a wrapper for the *exec* system call (to prevent its use), but not for the *fork* system call (which is by default allowed).

While there are good reasons to disallow system calls by default, protecting the "dangerous" calls and allowing all others is less likely to break existing code than the alternate approach.

```
  wrapper httpProxy {

    static int wrapperSetupAllowed;

    wr_activate () {
      // Prevents activation of other
      // wrappers once this one is started
      wrapperSetupAllowed = FALSE;
  }

    wr_duplicate () {
      // In case of fork(), prevents
      // child from replacing the wrapper
      wrapperSetupAllowed = FALSE;
    }

  WR_LOCAL::op{exec||execve}
    pre {
      wr_printf ("!!! Denying '%s'\n", $$);
      return WR_DENY | WR_BADPERM;
    };

  WR_LOCAL::op{bind}
    pre {
      struct   sockaddr_in   *sock   =   (struct
sockaddr_in *) $addr;

      if (sock->sin_port != BIND_PORT) {
        wr_printf ("!!! Denying access to port
%d\n", sock->sin_port);
        return WR_DENY | WR_BADPERM;
      }
    };

  WR_LOCAL::op{open || open64}
    pre {
      if (($flags & (O_RDWR | O_WRONLY)) == 0) {
        /* file is read only */
        if (wr_strcmp ($path,
              "/etc/.name_service_door")
              != 0 &&
            wr_strcmp ($path,
              "/etc/.syslog_door") != 0 &&
            wr_strcmp ($path,
              "/etc/netconfig") != 0 &&
            wr_strcmp ($path,
              "/etc/hosts") != 0 &&
            wr_strcmp ($path,
              "/etc/nsswitch.conf") != 0 &&
            wr_strcmp ($path, "/dev/zero")
              != 0 &&
            wr_strcmp ($path, "/dev/conslog")
              != 0 &&
            wr_strncmp ($path,
              "/opt/SUNWspro/lib", 16)
              != 0 &&
            wr_strncmp ($path,
              "/usr/platform", 13) != 0 &&
            wr_strncmp ($path,
              "/usr/local/etc", 14) != 0 &&
            wr_strncmp ($path, "/usr/lib", 8)
              != 0 &&
            wr_strncmp ($path,
              "/usr/share/lib", 14) != 0)
          {
            wr_printf ("!!! Denying open for
read of '%s'\n", $path);
            return WR_DENY | WR_BADPERM;
          }
        }
        else {
          /* open for write */
          if (wr_strcmp ($path,
                "/dev/conslog") != 0 &&
              wr_strcmp ($path, "/dev/null")
                != 0 &&
              wr_strncmp ($path,
                "/var/run/proxy.", 15) != 0)
          {
            wr_printf ("!!! Denying open for
writing of '%s'\n", $path);
            return WR_DENY | WR_BADPERM;
          }
        }
      };

  WR_LOCAL::oattr{fileop}
    pre {
      if (wr_strcmp ($$, "open")   != 0 &&
          wr_strcmp ($$, "open64") != 0 &&
          wr_strcmp ($$, "stat")   != 0) {
        wr_printf ("!!! Denying fileop '%s'\n",
$$);
        return WR_DENY | WR_BADPERM;
      }
    };

  WR_LOCAL::op{chroot}
    pre {
      if (wr_strcmp ($path, CHROOT_PATH) == 0) {
        wrapperSetupAllowed = TRUE;
      }
      else {
        wr_printf   ("!!!   Denying   chroot   to
'%s'\n", $path);
        return WR_DENY | WR_BADPERM;
      }
    };

  WR_LOCAL::op{setuid}
    pre {
      if ($uid == SETUID_ID) {
        wrapperSetupAllowed = TRUE;
      }
      else {
        wr_printf ("!!! Denying setuid to %d\n",
uid);
        return WR_DENY | WR_BADPERM;
      }
    };

  WR_LOCAL::op{setgid}
    pre {
      if ($gid == SETGID_ID) {
        wrapperSetupAllowed = TRUE;
      }
      else {
        wr_printf ("!!! Denying setgid to %d\n",
gid);
        return WR_DENY | WR_BADPERM;
```

```
    }
  };

// Catch all root operations and deny them
WR_LOCAL::oattr{rootop}
  pre {
    if (wrapperSetupAllowed)
      wrapperSetupAllowed = FALSE;
    else {
      return WR_DENY | WR_BADPERM;
    }
  };

}
```

Figure 3. HTTP Proxy Wrapper

The most interesting part of the wrapping exercise was testing. While we assume there are flaws in any proxy, we decided that it was easier (and perhaps more instructive), to deliberately introduce flaws, and then verify that the wrappers prevented a user from exploiting the flaws. Our main concern in deliberately introducing a flaw was the risk that it might somehow get checked in to the main source tree, and become part of the shipping source code. To avoid this, we have taken steps to ensure that the modified source code is not provided to anyone in the product development organization.

Our deliberate flaws included introducing a capability for creating a root shell and binding to a disallowed port. Because the *exec* system call was blocked by the wrapper, the effort to create the root shell failed. Because the *bind* system call was only allowed to the specified port, it failed as well.

As an adjunct to this effort, we developed a capability to generate alerts to an intrusion detection system [IDIP]. The concept is that when the wrapper detects an attempt to invoke a prohibited system call, this may indicate that a flaw has been found in the proxy and that the proxy is misbehaving. Of course, it may simply indicate that a code path is being exercised that was not found in developing the wrapper! Some system calls are more likely to indicate an attempted subversion than others. For example, for most proxies an *exec* call, an attempt to bind ports other than that assigned to the proxy, or an attempt to read `/etc/passwd` is a sure sign of a break-in attempt, while attempts to read other files may be more likely to indicate an error in the wrapper. Interfacing with the intrusion detection system was done by writing records to the log file, and having a user space daemon read the log file and forward the relevant records to the intrusion detection system for processing. Because of timing concerns, it was not feasible to directly invoke the intrusion detection system from the wrapper running in the kernel.

The Gauntlet wrappers themselves are quite small. Not including the wrappers libraries, each wrapper is about 250 lines of commented code, thus lending themselves to easy inspection.

A related effort involved wrapping the Multi-Protocol Object Gateway (MPOG) [Lamperillo], a gateway for CORBA, RMI, and DCOM traffic. Unlike traditional proxies, MPOG is written in Java, and runs under the Java Virtual Machine (JVM). Wrapping a JVM turned out to be difficult, because it performs many actions as part of startup that would ideally be blocked. The resulting wrapper, an excerpt of which is shown in Figure 4, is far more permissive than the corresponding HTTP proxy wrapper. We were quite surprised at some of the permissions necessary for proper operation of the JVM. For example, the JVM opens both `/dev/null` and `/dev/zero` for reading and writing. We would not have been surprised at opening `/dev/null` for writing, or `/dev/zero` for reading, but we were surprised that all combinations were required. We were disappointed that we had to open so many files for reading (e.g., /etc/hosts, /etc/resolv.conf) which may be of use to an attacker, should they be able to subvert the proxy. The number of files (and the specifics of the files) required by the JVM reinforced our belief that proxies should not be written in Java.

```
wrapper mpog {

WR_LOCAL::op{exec||exece}
  pre {
    wr_log ("%s(%d): !!! Denying '%s'\n",
      _progname, _pid, $$);
    return WR_DENY | WR_BADPERM;
  };

WR_LOCAL::oattr{forkop}
  pre {
    wr_log ("%s(%d): !!! Denying '%s'\n",
      _progname, _pid, $$);
    return WR_DENY | WR_BADPERM;
  };

WR_LOCAL::op{open || open64}
  pre {
    if (($flags & (O_RDWR | O_WRONLY)) == 0) {
      /* file is read only */
      if (wr_strcmp ($path,
            "/etc/.name_service_door") != 0 &&
          wr_strcmp ($path,
            "/etc/.syslog_door") != 0 &&
          wr_strcmp ($path,
            "/etc/netconfig") != 0 &&
          wr_strcmp ($path,
            "/etc/hosts") != 0 &&
          wr_strcmp ($path,
            "/etc/mnttab") != 0 &&
          wr_strcmp ($path,
            "/etc/protocols") != 0 &&
          wr_strcmp ($path,
            "/etc/resolv.conf") != 0 &&
          wr_strcmp ($path,
            "/etc/nsswitch.conf") != 0 &&
          wr_strcmp ($path,
            "/dev/zero") != 0 &&
          wr_strcmp ($path,
            "/dev/null") != 0 &&
```

```
            wr_strcmp ($path,
              "/dev/conslog") != 0 &&
            wr_strncmp ($path,
              "/opt/SUNWspro/lib", 16) != 0 &&
            wr_strncmp ($path,
              "/usr/platform", 13) != 0 &&
            wr_strncmp ($path,
              "/usr/local/etc", 14) != 0 &&
            wr_strncmp ($path,
              "/usr/java", 9) != 0 &&
            wr_strncmp ($path,
              "/usr/openwin", 12) != 0 &&
            wr_strncmp ($path,
              "/proc/", 6) != 0 &&
            wr_strncmp ($path, "./", 2) != 0 &&
            wr_strncmp ($path,
              "/usr/lib", 8) != 0 &&
            wr_strncmp ($path,
              "/export/home/guest/MPOG",  23)  !=
0 &&
            wr_strncmp ($path,
              "/usr/share/lib", 14) != 0)
        {
          wr_log ("%s(%d): !!! Denying open for
read of '%s'\n",
                  _progname, _pid, $path);
          return WR_DENY | WR_BADPERM;
        }
      }
      else {


        /* open for write */
        if (wr_strcmp ($path,
              "/dev/conslog") != 0 &&
            wr_strcmp ($path,
              "/dev/null") != 0 &&
             wr_strcmp ($path,
               "/dev/zero") != 0 &&
             wr_strcmp ($path,
               "/usr/local/etc/mp_orb_gw")
               != 0 &&
             wr_strncmp ($path,
               "/var/run/proxy.", 15) != 0)
        {
          wr_log ("%s(%d): !!! Denying open for
writing of '%s'\n",
                  _progname, _pid, $path);
          return WR_DENY | WR_BADPERM;
        }
      }
    };

  WR_LOCAL::oattr{fileop}
    pre {
      if (
        wr_strcmp ($$, "open")   != 0 &&
        wr_strcmp ($$, "open64") != 0 &&
        wr_strcmp ($$, "resolvepath") != 0 &&
        wr_strcmp ($$, "pathconf") != 0 &&
        wr_strcmp ($$, "stat64") != 0 &&
        wr_strcmp ($$, "lstat64") != 0 &&
        wr_strcmp ($$, "stat")   != 0)
      {
        wr_log  ("%s(%d):  !!!  Denying  fileop
'%s'\n", _progname, _pid, $$);
        return WR_DENY | WR_BADPERM;
      }
      else
        return WR_ALLOW;
      };

  WR_LOCAL::oattr{rootop}
    pre {
      wr_log ("%s(%d): !!! Denying call '%s'\n",
_progname, _pid, $$);
      return WR_DENY | WR_BADPERM;
    };

  }  // End of wrapper
```

Figure 4. MPOG Proxy Wrapper

5 Wrapping Non-source Firewalls Using NAI Labs Wrappers

In Section 4 we described how we wrapped the Gauntlet firewall. Our next effort was to develop a wrapper for a firewall for which we had no design information or source code. The goal was to see if the same techniques would work as they did for Gauntlet, and to determine whether we could add strength to competitive commercial products just as we did for Gauntlet.

At the time this effort began, NAI Labs wrappers were available with FreeBSD 2.2.7 and Solaris 2.6. A version for Windows NT was under development, but was not stable enough for use when we began this experiment. A market survey revealed four firewall products that run on Solaris 2.6, and none that run on FreeBSD. Of the four products, two vendors were unwilling to cooperate due to the competitive nature of Gauntlet with their products. We obtained both of the other products for our experiment.

Our results thus far have been discouraging. The first product we selected was SmallWorks NetGate. After installing and configuring NetGate, we realized that we had never specifically asked how the proxying is implemented. NetGate implements its proxies entirely within the kernel. Since our wrappers technology rely on intercepting system calls as they go from user state to system state, we were unable to develop any wrappers for it. Thus, there was no further effort on NetGate.

The second firewall we selected was Milkyway Networks SecurIT. As compared to Gauntlet, where each proxy implements its own access controls (and thus is wrapped separately), SecurIT uses a single program, *guardian*, to implement most of the security policies, including decision making based on IP addresses. Thus, a single wrapper could constrain many of the proxies. A second wrapper, which we called *secureitproxy*, serves to constrain many of the common proxies including *ftp*, *telnet*, and *http*. The operations allowed by the wrappers were roughly equivalent to those allowed for comparable Gauntlet proxies.

The result was that we were modestly successful in wrapping a firewall for which we did not have source

code. We would like to (but have not yet) develop a method for measuring how much resilience was added by the wrappers. Also, because we have much less familiarity with the SecurIT product than Gauntlet, we would like to exercise the system more thoroughly, to see whether our wrapper improperly prohibited any activities which should have been allowed. The gross differences in firewall architectures were disappointing, in that they limited the usefulness of our approach.

Files	Gauntlet UNIX	Gauntlet NT
Hosts file	/etc/hosts	$WINDIR\System32\drivers\etc\hosts
Services file	/etc/services	$WINDIR\System32\drivers\etc\services
Netperm-table	/usr/local/etc/netperm-table	C:\Program Files\Network Associates\Gauntlet\netperm-table
Log file	/dev/log	\\.\pipe\Gauntlet\LogsService

(Where "$WINDIR" is the Windows NT directory, generally "C:\WINNT")

Figure 5. Comparison of Gauntlet UNIX and NT File Restrictions.

6 Wrapping Gauntlet Using ISI Wrappers

Building on our results wrapping Gauntlet with NAI Labs wrappers, we decided to develop a set of wrappers for the Windows NT version of Gauntlet. For this effort, we used the ISI version of Wrappers [Balzer].

There are several key differences between NAI Labs' UNIX wrappers and ISI's NT wrappers. As noted in Section 3, ISI wrappers are implemented in user space, and as such can be bypassed by anyone who writes assembly or machine language code to make direct calls rather than going through the APIs. Thus, buffer overrun attacks will succeed against the ISI wrappers (since the overrun will typically operate by inserting machine code to perform its operations, which would not be stopped by the API-only view of the wrapper), while such attacks can be stopped by the NAI Labs' UNIX wrappers, which operate inside the kernel.[5]

ISI wrappers are implemented as Dynamic Link Libraries (DLLs), and do not have a language for specifying wrappers akin to WDL. Windows NT has roughly 10 times more system call interfaces than a typical UNIX system. The lack of a characterization capability in ISI wrappers, together with the ten-fold increase in the number of interfaces available, made the wrapper definition much more time consuming. Additionally, the number and type of interfaces provided by NT changes radically from one version to another. Hence, there can be little confidence that NT wrappers are as thorough as the UNIX wrappers in constraining the behavior of misbehaving applications.

As in the case of the Gauntlet UNIX wrapping (section 4) and SecurIT wrapping (section 5), we focused on wrapping a small number of proxies (specifically, the *ftp*, *http*, and *smtp* proxies). While the policies we attempted to enforce were reasonably similar to the other firewalls, the implementations were rather different due to the differences between NT and UNIX. As an example, Figure 5 shows a comparison of the file portion of the wrapper.

Similarly, while Gauntlet UNIX stores most of its configuration data in files, Gauntlet NT stores its data in the Windows Registry. Thus, the wrapper needs to allow access to those portions of the Registry needed by the proxy.

An area of difficulty was process creation. While UNIX systems have *fork* and *vfork* system calls, NT systems have approximately 50 different APIs that can be used to create a new process. We determined experimentally that they seem to all call the same entry point in the kernel, although it is difficult to know whether this is completely accurate. An attacker using a buffer overrun attack may cause execution of arbitrary machine instructions, which may invoke some of these alternate entry points (if they in fact exist). Since we can have no assurance that these alternate entry points are unused, it is safest to intercept them all. Also, Microsoft periodically changes the internal linkages between APIs and system calls, so there is no way to know whether this is a permanent situation. Thus, while we wrapped the single *CreateProcessW* API, we are unsure whether this would actually constrain a malicious proxy from creating other processes.

As noted above, each system call must be wrapped individually. Because of the lack of characterization, the wrapper developer must be intimately aware of the parameter types and orders, unlike with the NAI Labs wrapper technology. As a result, the development effort using ISI wrappers to wrap Gauntlet was much greater than using NAI Labs wrappers.

[5] We note that NAI Labs' NT wrappers, which are not discussed in this paper, are subject to the same vulnerability as the ISI NT wrappers.

7 Current Status and Future Directions

Wrappers are not a universal solution, even for firewalls. Wrappers cannot solve certain types of security flaws, such as:

- An HTTP proxy that aims to block URLs based on the name of the URL (e.g., one containing the string "sex") or the content of the page.[6] Without replicating the logic of the proxy (which would mean duplicating the effort and complexity of the proxy in the wrapper), it is infeasible to verify the correct operation of the proxy against such requirements.
- Proper implementation of an authentication method cannot be verified. For example, a wrapper could not protect against a malfunctioning *telnet* proxy that implements a one time password, but allows any users through even if the password does not match. Similarly, a wrapper cannot protect against a proxy that makes incorrect decisions based on SSL certificates.

In short, only those operations visible at the system call level with a minimum of understanding of the context can be protected using wrappers.

Our effort involved writing wrappers for firewalls that already exist, and in most cases do not have detailed designs. Thus, determining which system calls should be allowed and which should be rejected was a bottom-up effort, rather than top-down. We started by plugging in a logging wrapper which records each system call and its parameters, and then exercised the proxy. We undoubtedly did not exercise all call paths, and as a result it may be that the wrapper will refuse to allow certain (legal) operations that the proxy requires. This method of reverse engineering is time-consuming and error prone, but it is the only effective method given the lack of design information.

The integration of NAI Labs wrappers with Gauntlet has been successfully demonstrated in DARPA's Technology Integration Center (TIC) to DARPA program managers as well as VIPs from other parts of government. The additional strength gained from the wrappers is a good example of "defense in depth",[7] one of the key approaches being used to provide security within the Department of Defense.

As part of the TIC demonstrations, government-sponsored "red teams" have attempted to subvert the wrappers. In fact, one of our motivations in developing the wrappers was a red team exercise in which the attackers replaced the HTTP proxy with a booby-trapped copy which let them through at an appropriate time. Once we instituted the wrappers, the barrier was raised: the red team would have to not only replace the proxy, but also replace the wrapper with a corresponding wrapper that allowed the previously prohibited behavior to occur. We did not have the opportunity to run a second red team exercise after the wrappers were instituted, so we cannot determine how much additional resistance they provided.

The wrappers technology, including the firewall wrappers described in this paper can be downloaded from `ftp.tislabs.com/pub/wrappers` .

We have not performed any performance analysis of the wrapped firewalls. However, performance analysis of the base NAI Labs wrapper technology indicates that the load should be very small (*i.e.*, probably in the range of 3-5%).

8 Conclusion

Wrappers can provide the ability to constrain the behavior of a firewall proxy, just as they can constrain the behavior of any other program in a general purpose computing system. Given the premise that wrappers are simpler (and hence more readily understandable) than the programs they wrap, a wrapper for a proxy adds a layer of protection even in cases where source code is available. In cases where source code is not available, wrappers may provide the only meaningful way to constrain the behavior of a proxy. By adding wrappers to a security device such as a firewall, we gain confidence that the firewall cannot be subverted, even if flaws in the proxies might otherwise allow such subversion to occur. Operating systems with more granular controls would render this technology unnecessary.

As one of the reviewers of this paper wrote, "… wrappers add security functionality to an operating system which helps improve firewalls… [which] implies that operating system controls are not very helpful – after all, not much about them has changed in 40 years."

9 Acknowledgements

This project would not have occurred without the encouragement of Sami Saydjari at DARPA. We also appreciate the encouragement from our colleagues on the DARPA Information Assurance program, including John Lowry, Andy Thompson, David Levin, and Gregg Schudel (all from Verizon, neé BBN) and Dan Schnackenberg (from Boeing). Finally, we appreciate the contributions of our many colleagues at NAI Labs, including Terry Benzel, Dale Johnson, Dan Sterne, ad Lee Badger, Mark Feldman, and Doug Kilpatrick.

[6] Whether such "censorship" of URLs is good or bad, and the side effects of blind pattern matching, is beyond the scope of this paper.

[7] The use of wrappers together with proxies is different from other uses of defense in depth, where multiple independent systems are used (e.g., a firewall with an intrusion detection system). However, it is an equally valid use of the concept: both the proxy and the wrapper are seeking to prevent an intruder from getting in to the internal system or into the firewall itself.

We appreciate the comments from the anonymous reviewers, who helped improve this paper.

10 References

[Balzer] Robert Balzer and Neil Goldman, "Mediating Connectors", *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop*, Austin, TX May 31 - June 5, 1999.

[Fraser] Timothy Fraser, Lee Badger, and Mark Feldman, "Hardening COTS Software with Generic Software Wrappers", *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland CA, May 1999.

[IDIP] "Dynamic, Cooperating Boundary Controllers Final Technical Report", Boeing report D658-10822-1, August 1998.

[Jones] Michael Jones, “Interposition Agents: Transparently Interposing User Code at the System Interface”, ACM Symposium on Operating System Principles, 1993.

[Lamperillo] Gary Lamperillo, "Architecture and Concepts of the MPOG", submitted for publication.

[Venema] Wietse Venema, “TCP Wrapper: Network Monitoring, Access Control, and Boob Ttraps”, *Proceedings of the 3rd UNIX Security Symposium*, Baltimore MD, September 1992.