Non-Control-Data Attacks Are Realistic Threats

Shuo Chen[†], Jun Xu[‡], Emre C. Sezer[‡], Prachi Gauriar[‡], and Ravishankar K. Iyer[†]

[†] *Center for Reliable and High Performance Computing,*
Coordinated Science Laboratory,
University of Illinois at Urbana-Champaign,
1308 W. Main Street, Urbana, IL 61801
{shuochen,iyer}@crhc.uiuc.edu

[‡] *Department of Computer Science*
North Carolina State University
Raleigh, NC 27695
{jxu3,ecsezer,pgauria}@ncsu.edu

Abstract

Most memory corruption attacks and Internet worms follow a familiar pattern known as the *control-data attack*. Hence, many defensive techniques are designed to protect program control flow integrity. Although earlier work did suggest the existence of attacks that do not alter control flow, such attacks are generally believed to be rare against real-world software. The key contribution of this paper is to show that non-control-data attacks are realistic. We demonstrate that many real-world applications, including FTP, SSH, Telnet, and HTTP servers, are vulnerable to such attacks. In each case, the generated attack results in a security compromise equivalent to that due to the control-data attack exploiting the same security bug. Non-control-data attacks corrupt a variety of application data including user identity data, configuration data, user input data, and decision-making data. The success of these attacks and the variety of applications and target data suggest that potential attack patterns are diverse. Attackers are currently focused on control-data attacks, but it is clear that when control flow protection techniques shut them down, they have incentives to study and employ non-control-data attacks. This paper emphasizes the importance of future research efforts to address this realistic threat.

1 Introduction

Cyber attacks against all Internet-connected computer systems, including those in critical infrastructure, have become relentless. Malicious attackers often break into computer systems by exploiting security vulnerabilities due to low-level memory corruption errors, e.g., buffer overflow, format string vulnerability, integer overflow, and double free. These vulnerabilities not only are exploited by individual intruders, but also make systems susceptible to Internet worms and distributed denial of service (DDoS) attacks. Recipe-like attack-construction documents [2][46] widely available on the Internet have made this type of attack widely understood.

Most memory corruption attacks follow a similar pattern known as the *control-data attack*: they alter the target program's control data (data that are loaded to processor program counter at some point in program execution, e.g., return addresses and function pointers) in order to execute injected malicious code or out-of-context library code (in particular, return-to-library attacks). The attacks usually make system calls (e.g., starting a shell) with the privilege of the victim process. A quick survey of the CERT/US-CERT security advisories [11][47] and the Microsoft Security Bulletin [26] shows that control-data attacks are considered the most critical security threats.

Because control-data attacks are currently dominant, many defensive techniques have been proposed against such attacks. It is reasonable to ask whether the current dominance of control-data attacks is due to an attacker's inability to mount non-control-data attacks[1] against real-world software. We suspect that attackers may in general be capable of mounting non-control-data attacks but simply lack the incentive to do so, because control-data attacks are generally easier to construct and require little application-specific knowledge on the attacker's side. If this is indeed true, when the deployment of control flow protection techniques makes control-data attacks impossible, attackers may have the incentive to bypass these defenses using non-control-data attacks.

The emphasis of this paper is the *viability* of non-control-data attacks against *real-world* applications. The possibility of these attacks has been suggested in previous work [9][42][48][52]. However, the applicability of these attacks has not been extensively studied, so it is not clear how realistic they are against real-world applications.

[1] Other terms are used to refer to attacks that do not alter control flow. For example, Pincus and Baker call them *pure data exploits* [29]. We call them *non-control-data attacks* mainly to contrast with control-data attacks.

The contribution of this paper is to experimentally demonstrate that non-control-data attacks are realistic and can generally target real-world applications. The target applications are selected from the leading categories of vulnerable programs reported by CERT from 2000 to 2004 [11], including various server implementations for the HTTP, FTP, SSH, and Telnet protocols. The demonstrated attacks exploit buffer overflow, heap corruption, format string, and integer overflow vulnerabilities. All the non-control-data attacks that we constructed result in security compromises that are as severe as those due to traditional control-data attacks — gaining the privilege of the victim process. Furthermore, the diversity of application data being attacked, including configuration data, user identity data, user input data, and decision-making data, shows that attack patterns can be very diverse.

The results of our experiments show that attackers can indeed compromise many real-world applications without breaking their control flow integrity. We discuss the implications of this finding for a broad range of security defensive techniques. Our analysis shows that finding a generic and secure solution to defeating memory corruption attacks is still an open problem when non-control-data attacks are considered. Many available defensive techniques are not designed for such attacks: some address specific types of memory vulnerabilities, such as *StackGuard* [14], *Libsafe* [7] and *FormatGuard* [8]; some have practical constraints in the secure deployments, such as pointer protection [9] and address-space randomization [4][6]; and others rely on control flow integrity for security, such as system call based intrusion detection techniques [17][18][19][21][22][23][34][47], control data protection techniques [10][35][42], and non-executable-memory-based protections [1][41].

In addition to demonstrating the general applicability of non-control-data attacks, this paper can also be viewed as a step toward a more systematic approach to the empirical evaluation of defensive techniques. With more and more promising defensive techniques being proposed, researchers have started to realize the necessity of empirical evaluation. In a survey paper [29], Pincus and Baker explicitly call for a thorough study of whether current defensive techniques "give sufficient protection in practice that exploitation of low-level defects will cease to be a significant elevation of privilege threat."

The rest of the paper is organized as follows: Section 2 discusses the motivation for examining the applicability of non-control-data attacks. Section 3 and 4 present our experimental work on constructing attacks by tampering with many types of security-critical data other than control data. In Section 5, the results of the experiments are used to re-examine the effectiveness of a number of security defensive methods. The constraints and counter-measures of non-control-data attacks are discussed in Section 6. We present related work in Section 7 and conclude with Section 8.

2 Motivation

While control-data attacks are well studied and widely used, the current understanding of non-control-data attacks is limited. Although their existence has been known (e.g., Young and McHugh [52] gave an example of such attacks in a paper published even before the spread of the notorious Morris Worm[2]), the extent to which they are applicable to real-world applications has not been assessed. Because non-control-data attacks must rely on specific semantics of the target applications (e.g., data layout, code structure), their applicability is difficult to estimate without a thorough study of real vulnerabilities and the corresponding application source code. Control-data attacks, on the other hand, are easily applicable to most real-world applications once the memory vulnerabilities are discovered.

This paper is also motivated by results from a number of research papers investigating the impact of random hardware transient errors on system security. Boneh et al. [5] show that hardware faults can subvert an RSA implementation. Our earlier papers [15][50] indicate that even random memory bit-flips in applications can lead to serious security compromises in network servers and firewall functionalities, These bit-flip-caused errors include corrupting Boolean values, omitting variable initializations, incorrect computation of address offsets and corrupting security rule data. Govindavajhala and Appel conduct a physical random fault injection experiment to subvert the Java language type system [20]. All these security compromises are very specific to application semantics, and not due to control flow altering. It should be noted, however, that the security compromises caused by hardware faults only suggest potential security threats, since attackers usually do not have the power to inject physical hardware faults to the target systems. Nevertheless, the most compelling message from these papers is that real-world software applications are very likely to contain security-critical non-control data, given that *even random hardware errors can hit them with a non-negligible probability*.

[2] One of the attack vectors of the Morris Worm overruns a stack buffer in *fingerd* to corrupt a return address. This worm made control-data attacks widely known to the public.

We realize that several types of memory corruption vulnerabilities, in particular, format string vulnerability, heap overflow, signed integer overflow, and double free vulnerabilities, are essentially memory fault injectors: they allow attackers to overwrite arbitrary memory locations within the address space of a vulnerable application. Compared to hardware transient errors, software vulnerabilities are more deterministic in that they always occur in the programs, They are also more amenable to attacks in that target memory locations can be precisely specified by the attacker. Based on these observations, we make the following claim:

> **Applicability Claim of Non-Control-Data Attacks:** *Many real-world software applications are susceptible to non-control-data attacks, and the severity of the resulting security compromises is equivalent to that of control-data attacks.*

Since this is a claim about real-world software, we selected a number of representative applications and constructed non-control-data attacks in order to answer three major questions: (1) Which data within the target applications are critical to security other than control data? (2) Do the vulnerabilities exist at appropriate stages of the application's execution that can lead to eventual security compromises? (3) Is the severity of the security compromises equivalent to that of traditional control-data attacks?

3 Security-Critical Non-Control Data

In preparation for the proposed experiment, we studied several network server applications, and experimented with many types of non-control data. The study showed that the following types of data are critical to software security:

- Configuration data
- User input
- User identity data
- Decision-making data

These classes are not meant to be mutually exclusive or collectively complete, but rather, the classification organizes reasoning about possibilities of non-control-data attacks. In this section, we explain each of these data types and why each is critical to security. For each data type, we describe the attack scheme(s) in Section 4 using real-world applications.

As indicated earlier, identifying security-critical non-control data and constructing corresponding attacks require sophisticated knowledge about program semantics. We currently rely on manual analysis of source code to obtain such knowledge.

Configuration Data. Site-specific configuration files are widely used by many applications. For example, many settings of the Apache web server can be configured by the system administrator using *httpd.conf*. The administrator can specify locations of data and executable files, access control policies for the files and directories, and other security and performance related parameters [3]. Similar files are used by FTP, SSH, and other network server applications. Usually, the server application processes the configuration files to initialize internal data structures at the very beginning of program execution. At runtime, these data structures are used to control the behaviors of the application, and they rarely change once the server enters the service loop. Corrupting configuration data structures allows the attacker to change and even control the behaviors of the target application. In our study, we have focused on the file path configuration information. The file path directives define where certain data and executable files are located so that the server can find them at runtime. They also serve as access control policies. In the case of a web server, the CGI-BIN path directive is not only used to locate the CGI programs, but it also prevents a malicious client from invoking arbitrary programs, i.e., only a pre-selected list of trusted programs in the specified directory can be executed. If the configuration data can be overwritten through memory corruption vulnerabilities, an attacker can bypass the access control policy defined by the administrator.

User Identity Data. Server applications usually require remote user authentication before granting access. These privileged applications usually cache user-identity information such as user ID, group ID, and access rights in memory while executing the authentication protocol. The cached information is subsequently used by the server for remote access decisions. If the cached information can be overwritten in the window between the time the information is first stored in memory and the time it is used for access control, the attacker can potentially change the identity and perform otherwise unauthorized operations within the target system.

User Input String. Changing user input is another way to launch a successful non-control-data attack. Input validation is a critical step in many applications to guarantee intended security policies. If user input can be altered after the validation step, an attacker would be able to break into a system. We use the following steps in the attack: (1) first, use a legitimate input to pass the input validation checking in the application; (2) then, alter the buffered input data to become malicious; (3) finally, force the application to use the altered data. The attack described here is actually a type of TOCTTOU (Time Of Check To Time Of Use) attack: using legitimate data to pass the security checkpoint and then forcing the application to use corrupted data that it considers legitimate. In the existing literature, TOCTTOU is mainly described in the context of file race condition attacks. The attack studied here shows that the notion is applicable to memory data corruption as well.

Decision-Making Data. Network server applications usually use multiple steps for user authentication. Decision-making routines rely on several Boolean variables (conjunction, disjunction, or combination of both) to reach the final verdict. No matter how many steps are involved in the authentication, eventually at a single point in the program control flow, there has to be a conditional branch instruction saying either yes or no to the remote client. Although such a critical conditional branch instruction may appear in different places in the binary code, it nonetheless makes the critical decision based on a single register or memory data value. An attacker can corrupt the values of these final decision-making data (usually just a Boolean variable) to influence the eventual critical decision.

Other Non-Control Data for Future Investigation. We have discussed four different types of data that, if corrupted, can compromise security. Many other types of data are also critical to program security. We identify some of them for future investigation. File descriptors are integers to index the kernel table of opened files. They can point to regular disk files, standard input/output, and network sockets. If the attacker can change the file descriptors, the security of file system related operations can be compromised. Changing a file descriptor to that of a regular disk file could redirect terminal output to the file and result in severe security damage. Another possible target is the RPC (Remote Procedure Call) routine number. Each RPC service is registered with an integer as its index in the RPC callout link list. The caller invokes a service routine by providing its index. Malicious changes of RPC routine numbers could change the program semantics without running any external code.

4 Validating the Applicability Claim

In this section we validate the Applicability Claim, stated in Section 2. It would be straightforward to manually construct vulnerable code snippets to demonstrate non-control-data attacks. This, however, does not validate the claim because what we need to show is the applicability of such attacks on a variety of real-world software applications. Toward this end, we need first to understand which applications are frequent targets of attacks and what types of vulnerabilities are exploited. A quick survey was performed on all 126 CERT security advisories between the years 2000 and 2004. There are 87 memory corruption vulnerabilities, including buffer overflow, format string vulnerabilities, multiple free, and integer overflow. We found that 73 of them are in applications providing remote services. Among them, there are 13 HTTP server vulnerabilities (18%), 7 database service vulnerabilities (10%), 6 remote login service vulnerabilities (8%), 4 mail service vulnerabilities (5%), and 3 FTP service vulnerabilities (4%). They collectively account for nearly half of all the server vulnerabilities.

Our criteria in selecting vulnerable applications for experimentation are as follows: (1) Different types of vulnerabilities should be covered. (2) Different types of server applications should be studied in order to show the general applicability of non-control-data attacks, (3) There should be sufficient details about the vulnerabilities so that we can construct attacks based on them. There are a number of practical constraints and difficulties in this enterprise. A significant number of vulnerability reports do not claim with certainty that the vulnerabilities are actually exploitable. Among the ones that do, many do not provide sufficient details for us to reproduce them. A number of the vulnerabilities that do meet our criteria are in proprietary applications. Therefore, we used open-source server applications for which both source code and detailed information about the vulnerabilities are available.

The rest of this section presents experimental results. The demonstrated non-control-data attacks can be categorized along two dimensions: the type of security-critical data presented in Section 3 and the type of memory errors, such as buffer overflow and format string vulnerability. Although a significant portion of this section is intended to illustrate various individual non-control-data attacks in substantial detail, the goal is to validate the applicability claim stated earlier.

4.1 Format String Attack against User Identity Data

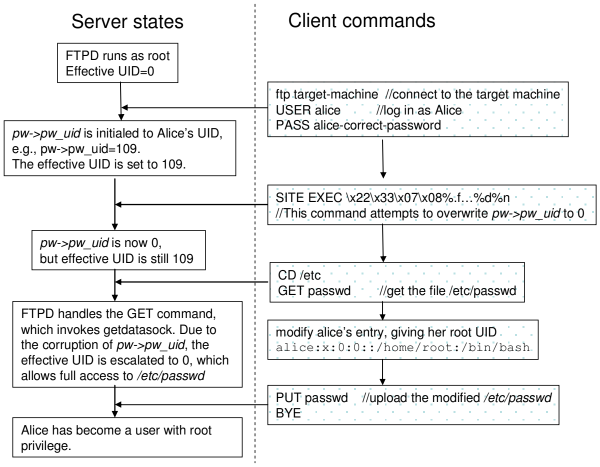
WU-FTPD is one of the most widely used FTP servers. The *Site Exec Command Format String Vulnerability* [12] is one that can result in malicious code execution with root privilege. All the attack programs we obtained from the Internet overwrite return addresses or function pointers to execute a remote root shell.

Our goal is to construct an attack against user identity data that can lead to root privilege compromise without injecting any external code. Our first attempt was to find data items that, if corrupted, could allow the attacker to log in to the system as root user without providing a correct password. We did not succeed in this because the SITE EXEC format string vulnerability occurs in a procedure that can only be invoked after a successful user login. That means an attacker could not change data that would directly compromise the existing authentication steps in FTPD. Our next attempt was to explore the possibility of overwriting the information source that is used for authentication. In UNIX-based systems, user names and user IDs are saved in a file called */etc/passwd*, which is only writable to a privileged root user. A natural thought is to corrupt information in this file in order to get into the system. By overwriting an entry in this file, an attacker can later legitimately log in to the victim machine as a privileged user. We observed that after a successful user login, the effective UID (EUID) of the FTPD process has properly dropped to the user's UID, so the process runs as an unprivileged user. Therefore, */etc/passwd* can be overwritten only if we can escalate the privilege of the server process to root privilege. This is possible because the real UID of the process is still 0 (root UID) even after its EUID is set to be the user's UID. The success of the attack depends on whether we can corrupt a certain data structure so that the EUID can be reverted to 0. FTPD uses the *seteuid()* system call to change its EUID when necessary. There are 18 *seteuid(0)* invocations in the WU-FTPD source code, one of which appears in function *getdatasock()* shown in Table 1. The function is invoked when a user issues data transfer commands, such as *get* (download file) and *put* (upload file). It temporarily escalates its privilege to root using *seteuid(0)* in order to perform the *setsockopt()* operation. It then calls *seteuid(pw->pw_uid)* to drop its privilege. The data structure *pw->pw_uid* is a cached copy of the user ID saved on the heap. Our attack exploits the format string vulnerability to change *pw->pw_uid* to 0, effectively disabling the server's ability to drop privilege after it is escalated. Once this is done, the remote attacker can download and upload arbitrary files from/to the server as a privileged user. The attack compromises the root privilege of FTPD without diverting its control flow to execute any malicious code.

Table 1: Source Code of `getdatasock()`

```
FILE * getdatasock( … ) {
  ...
  seteuid(0);
  setsockopt( ... );
  ...
  seteuid(pw->pw_uid);
  ...
}
```

The attack has been successfully tested on WU-FTPD-2.6.0. First we establish a connection to the control port of FTPD and correctly log in as a regular user, Alice. FTPD sets its effective user ID to that of Alice (e.g., 109). The client then sends a specially constructed SITE EXEC command to exploit the format string vulnerability that overwrites the *pw->pw_uid* memory word to 0. The client then establishes the data connection and issues a *get* command, which invokes the function *getdatasock()*. Due to the corruption of *pw->pw_uid*, the execution of the function sets the EUID of the process to 0, permanently. The client can therefore download */etc/passwd* from the server, add any entry desired, and then upload the file to the attacked server. An entry such as "*alice:x:0:0::/home/root:/bin/bash*" indicates that Alice can log in to the server as a root user anytime via FTP, SSH, or other available service. Figure 1 gives the state transition and flowchart of the attack.



FTPD runs as root
Effective UID=0

pw->pw_uid is initialed to Alice's UID,
e.g., pw->pw_uid=109.
The effective UID is set to 109.

pw->pw_uid is now 0,
but effective UID is still 109

FTPD handles the GET command, which invokes getdatasock. Due to the corruption of *pw->pw_uid*, the effective UID is escalated to 0, which allows full access to */etc/passwd*

Alice has become a user with root privilege.

```
ftp target-machine  //connect to the target machine
USER alice          //log in as Alice
PASS alice-correct-password
```

```
SITE EXEC \x22\x33\x07\x08%.f…%d%n
//This command attempts to overwrite pw->pw_uid to 0
```

```
CD /etc
GET passwd      //get the file /etc/passwd
```

```
modify alice's entry, giving her root UID
alice:x:0:0::/home/root:/bin/bash
```

```
PUT passwd    //upload the modified /etc/passwd
BYE
```

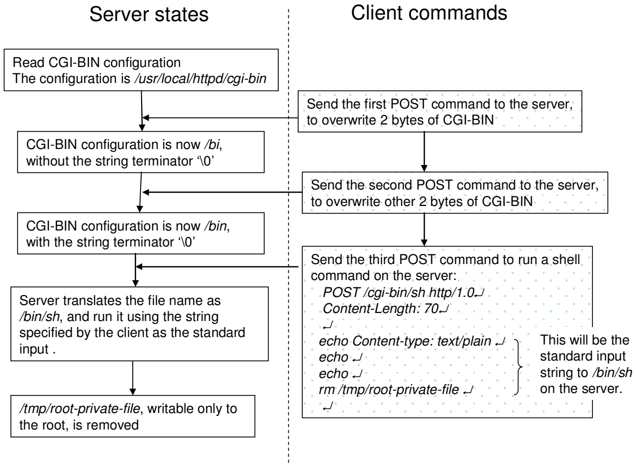
Figure 1: User Identity Data Attack against WU-FTPD

4.2 Heap Corruption Attacks against Configuration Data

Memory corruption vulnerabilities on an HTTP daemon and a Telnet daemon allow configuration data attacks to succeed in getting root shells, if these daemons run as root. Note that some HTTP daemons can run as an unprivileged user, e.g., a special user *nobody*, in which case the root compromise is unlikely to happen whether the attack is a control-data attack or a non-control data attack. Our applicability claim still stands, because the claim is that non-control-data attacks can get the same privilege level as control-data attacks, which is the privilege level of the victim server.

Attacking *Null HTTPD*. *Null HTTPD* is a multi-threaded web server on Linux. Two heap overflow vulnerabilities have been reported [38]. Available exploit programs overwrite a Global Offset Table[3] (GOT) entry of a function when the corrupted heap buffer is freed. The program control jumps to the attacker's malicious code when a subsequent invocation of the function is made.

It can be seen that corrupting the CGI-BIN configuration string can result in root compromise without executing any external code. CGI (Common Gateway Interface) is a standard for running executables on the server for data processing. As explain in Section 3, the CGI-BIN directive restricts a user from executing programs outside the CGI-BIN directory and is thus critical to the security of the HTTP server. A client's URL requesting the execution of a CGI program is always relative to the CGI-BIN configuration. Assuming the CGI-BIN path of the server *www.foo.com* is */usr/local/httpd/cgi-bin*, when a request of URL *http://www.foo.com/cgi-bin/bar* is processed, the HTTP server prefixes the CGI-BIN to *bar* and executes the file */usr/local/httpd/cgi-bin/bar* on the server's file system. Figure 2 shows our attack process, which overwrites the CGI-BIN configuration so that the shell program */bin/sh* can be started as a CGI program.


Server states
Client commands
Read CGI-BIN configuration
The configuration is /usr/local/httpd/cgi-bin
Send the first POST command to the server, to overwrite 2 bytes of CGI-BIN
CGI-BIN configuration is now /bi, without the string terminator '\0'
Send the second POST command to the server, to overwrite other 2 bytes of CGI-BIN
CGI-BIN configuration is now /bin, with the string terminator '\0'
Send the third POST command to run a shell command on the server:
Server translates the file name as /bin/sh, and run it using the string specified by the client as the standard input .
This will be the standard input string to /bin/sh on the server.
/tmp/root-private-file, writable only to the root, is removed

```
POST /cgi-bin/sh http/1.0↵
Content-Length: 70↵
↵
echo Content-type: text/plain ↵
echo ↵
echo ↵
rm /tmp/root-private-file ↵
↵
```

Figure 2: Configuration Data Attack against NULL HTTPD

The heap overflow vulnerability is triggered when a special POST command is received by the server. Due to the nature of heap corruption vulnerability, an attacker usually can only precisely control the first two bytes[4] in the corrupted word at a time to avoid a segmentation fault. We issue two POST commands to precisely overwrite four characters in the CGI-BIN configuration so that it is changed from "*/usr/local/httpd/cgi-bin\0*" to "*/bin\0*". After the corruption, we can start */bin/sh* as a CGI program and send any shell command as the standard input to */bin/sh*. For example, by issuing a *rm /tmp/root-private-file* command, we observe that the file */tmp/root-private-file*, writable only to root, was removed. This indicates that we are indeed able to run any shell command as root, i.e., the attack causes the root compromise.

Attacking *NetKit Telnetd*. A heap overflow vulnerability exists in many Telnet daemons derived from the BSD Telnet daemon, including a default RedHat Linux daemon *NetKit Telnetd* [13][39]. The vulnerability is triggered when the function *telrcv()* processes client requests of '*AYT*' (i.e., *Are-You-There*) configuration. The attack, downloaded from *Bugtraq,* overwrites a GOT entry to run typical malicious code starting a root shell.

When the daemon accepts a connection from a Telnet client, it starts a child process to perform user authentication. The file name of the executable for the authentication is specified by a configuration string *loginprg*, whose value can be specified as a command line argument. A typical value is */bin/login*. Suppose the remote user is from *attacker.com*. Function *start_login(host),* shown in Table 2, starts the command */bin/login –h attacker.com –p* by making an *execv* call to authenticate the user. The integrity of *loginprg* is critical to security.

Table 2: Attacking *loginprg* and *host* Variables in Telnet Daemon

```
void start_login(char * host,…) {
   addarg(&argv, loginprg);
   addarg(&arg, ″-h″);
   addarg(&argv, host);
   addarg(&arg, ″-p″);
   execv(loginprg, argv);
}
```

Without the corruption, the `execv` call is:

```
    /bin/login -h attacker.com -p
```

Due to the corruption, the `execv` call is:

```
  /bin/sh -h -p -p
```

We observe that the vulnerable function *telrcv()* can be invoked after the initializations of *loginprg* and *host* variables but before the invocation of

[3] The Global Offset Table (GOT) is a table of function pointers for calling dynamically linked library functions.

[4] If the value to be written is a valid address, four bytes can be overwritten by a single heap corruption attack.

start_login(host). Therefore, the exploitation of the heap overflow vulnerability allows overwriting the *loginprg* value to */bin/sh* and the *host* value to *–p*, so that the command */bin/sh –h –p –p* will be executed by function *start_login()*, giving a root shell to the attacker. Note that if *host* was not overwritten or if it was overwritten to an empty string, the *sh* command would generate a "file does not exist" error.

4.3 Stack Buffer Overflow Attack against User Input Data

Another HTTP server, GHTTPD, has a stack buffer overflow vulnerability in its logging function [36]. Unlike the heap corruption, integer overflow, or format string vulnerabilities, a stack overflow does not allow corruption of arbitrary memory locations but only of the memory locations following the unchecked buffer on the stack. The most popular way to exploit the stack buffer overflow vulnerability is to use the stack-smashing method, which overwrites a return address [2]. The attack overwrites the function return address saved on stack and changes it to the address of the injected malicious code, which is also saved in the unchecked buffer. When the function returns, it begins to execute the injected code. Stack buffer overflow attacks have been extensively studied, and many runtime protection solutions have been proposed. Most of the techniques try to detect corruption of return addresses. We construct an attack that neither injects code nor alters the return address. The attack alters only the backup value of a register in the function frame of the vulnerable function to compromise the security validation checks and eventually cause the root compromise.

The stack buffer overflow vulnerability is in function *log()*, where a long user input string can overrun a 200-byte stack buffer. A natural way to conduct a non-control-data attack is to see if any local stack variable can be overwritten. We were not able to find any local variable that can be used to compromise its security. Instead, we found that three registers from the caller were saved on the stack at the entry of function *log()* and restored before it returns. Register *ESI* holds the value of the variable *ptr* of the caller function *serveconnection()*. Variable *ptr* is a pointer to the text string of the URL requested by the remote client. Function *serveconnection()* checks if the substring "/.." (i.e., the parent directory) is embedded in the requested URL. Without the check, a client could execute *www.foo.com/cgi-bin/../bar*, an executable outside the restricted CGI-BIN directory. We observe that the function *log()* is called after *serveconnection()* checks the absence of "/.." in the URL, but before the CGI request is parsed and handled. This makes a TOCTTOU (Time Of Check To Time Of Use) attack possible. We first present a legitimate URL without "/.." to bypass the absence check, then we change the value of register *ESI* (value of *ptr*) to point to a URL containing "/.." before the CGI request is processed.

Table 3: Source Code of `servconnection()` and `log()`

```
int serveconnection(int sockfd) {
   char *ptr;  // pointer to the URL.
               // ESI is allocated
               // to this variable.
   ...
1: if (strstr(ptr,"/.."))
     reject the request;
2: log(...);
3: if (strstr(ptr,"cgi-bin"))
4:     Handle CGI request
   ...
}
```

```
Assembly of log(...)
   push %ebp
   mov %esp, %ebp
   push %edi
   push %esi
   push %ebx
   ... stack buffer overflow code
   pop %ebx
   pop %esi
   pop %edi
   pop %ebp
   ret
```

The attack scheme is given in Figure 3. The default configuration of GHTTPD is */usr/local/ghttpd/cgi-bin*, so the path */cgi-bin/../../../../bin/sh* is effectively the absolute path */bin/sh* on the server. We use the *GET* command of the HTTP protocol to trigger the buffer overflow condition and force the server to run */bin/sh* as a CGI program: we send the command "*GET AA…AA\xdc\xd7\xff\xbf↵↵/cgi-bin/../../../../bin/sh*" [5] to the server. The server converts the first part of the command, "*AAA…AAA\xdc\xd7\xff\xbf*", into a null-terminated string pointed to by *ptr* in function *serveconnection()*. This string passes the "/.." absence check in Line 1 of *serveconnection()*. When the string is passed to the *log()* function in Line 2, it overruns the buffer and changes the saved copy of register *ESI* (i.e., *ptr)* on the stack frame of *log()* to *0xbfffd7dc* (i.e., the bytes following "A" characters in the request), which is the address of the second part of the GET command "*/cgi-bin/../../../../bin/sh*". When *log()* returns, the value of *ptr* points to this unchecked string, which is a CGI request containing "/..". Succeeding in the check of

[5] "AA…AA" represents a long string of "A" characters.

Line 3, the request eventually starts the execution of */bin/sh* at Line 4 under the root privilege.

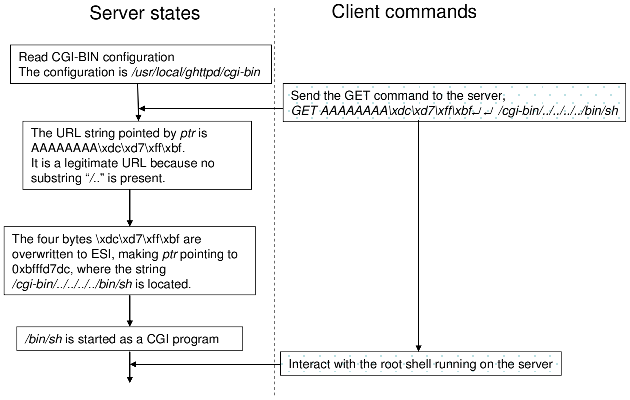

Server states
Client commands
Read CGI-BIN configuration
The configuration is /usr/local/ghttpd/cgi-bin
Send the GET command to the server,
GET AAAAAAAA\xdc\xd7\xff\xbf↵↵ /cgi-bin/../../../../bin/sh
The URL string pointed by ptr is
AAAAAAAA\xdc\xd7\xff\xbf.
It is a legitimate URL because no
substring "/.." is present.
The four bytes \xdc\xd7\xff\xbf are
overwritten to ESI, making ptr pointing to
0xbfffd7dc, where the string
/cgi-bin/../../../../bin/sh is located.
/bin/sh is started as a CGI program
Interact with the root shell running on the server

Figure 3: User Input Data Attack against GHTTPD

4.4 Integer Overflow Attack against Decision-Making Data

We also study decision-making data used by security-related operations in server applications. These data are usually Boolean variables used to see whether certain criteria are met by a remote client. If so, access will be granted. An attacker can exploit security vulnerabilities in a program to overwrite such Boolean variables and get access to the target system. We study the attack in the context of a secure shell (SSH) server implementation.

An integer overflow vulnerability [37] exists in multiple SSH server implementations, including one from SSH Communications Inc. and one from OpenSSH.org. The vulnerability is triggered when an extraordinarily large encrypted SSH packet is sent to the server. The server copies a 32-bit integer packet size value to a 16-bit integer. The 16-bit integer can be set to zero when the packet is large enough. Due to this condition, an arbitrary memory location can be overwritten by the attacker. Available exploitation online changes a function return address to run malicious shell code [37]. Detailed descriptions and analyses of this vulnerability can be found in [31] and [32].

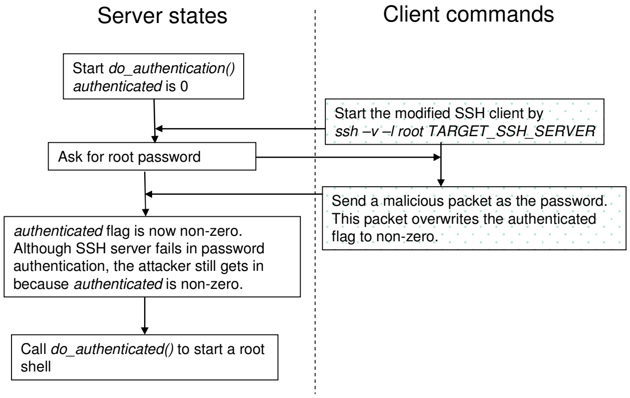
Our goal is to corrupt non-control data in order to log in to the system as root without providing a correct password. Our close examination of the source code of the SSH server implementation from SSH Communications Inc shows that the integer overflow vulnerability is in function *detect_attack()*, which detects the CRC32 compensation *attack* against the SSH1 protocol. This function is invoked whenever an encrypted packet arrives, including the encrypted user password packet. The SSH server relies on function *do_authentication()* (shown in Table 4) to authenticate remote users. It uses a *while* loop (line 2) to authenticate a user based on various authentication mechanisms, including *Kerberos* and *password*. The authentication succeeds if it passes any one of the mechanisms. A stack variable *authenticated* is defined as a Boolean flag to indicate whether the user has passed one of the mechanisms. The initial value of *authenticated* is 0 (i.e., false). Line 3 reads input packet using *packet_read()*, which internally invokes the vulnerable function *detect_attack()*. Our attack is to corrupt the *authenticated* flag and force the program to break out of the *while* loop and go to line 9, where a shell is started for the authenticated user.

Table 4: Source Code of `do_authentication()`

```
void do_authentication(char *user, ...) {
1:  int authenticated = 0;
   ...
2:  while (!authenticated) {
      /* Get a packet from the client */
3:    type = packet_read();
      // calls detect_attack() internally
4:    switch (type) {
      ...
5:    case SSH_CMSG_AUTH_PASSWORD:
6:     if (auth_password(user, password))
7:          authenticated =1;
      case ...
      }
8:    if (authenticated) break;
    }
    /* Perform session preparation. */
9:  do_authenticated(pw);
}
```

Our attack tries to log in as root without providing a correct password. When the server is ready to accept the root password, the SSH client sends a very large packet to the receiving function *packet_read()* (Line 3). The packet is specially formulated to trigger the integer overflow vulnerability when *packet_read()* calls *detect_attack()* for detection. As a result, the *authenticated* flag is changed to non-zero. Although the server does fail in function *auth_password()* (Line 6), it breaks out of the *while* loop and proceeds to create a shell for the client (Line 9). The client program successfully gets into the system without providing any password. Figure 4 shows the status of both the client and the server during the attack.

Currently our attack program has not calculated the correct checksum of the malicious packet that we sent to the server, so the packet would be rejected by the checksum validation code in the SSH server. For a proof-of-concept attack, we deliberately make the server accept the malicious packet without validating its checksum. To make the attack complete, we will need to understand the DES cryptographic algorithms to recalculate the checksum. Note that an attack including the checksum calculation algorithm is publicly available [32]. Other than this peculiarity, we have confirmed that the vulnerability allows precise corruption of the *authenticated* flag and that this corruption is sufficient to grant the root privilege to the attacker.



Start *do_authentication()*
authenticated is 0

Start the modified SSH client by
ssh –v –l root TARGET_SSH_SERVER

Ask for root password

Send a malicious packet as the password. This packet overwrites the authenticated flag to non-zero.

authenticated flag is now non-zero. Although SSH server fails in password authentication, the attacker still gets in because *authenticated* is non-zero.

Call *do_authenticated()* to start a root shell

Figure 4: Attacking Stack Variable *authenticated* in SSH Server

5 Implications for Defensive Techniques

The success in constructing non-control-data attacks for various network server applications suggest a re-examination of many current defensive techniques, which can be broadly categorized into two classes: techniques to avoid having memory-safety bugs in software and techniques to defeat exploitations of these bugs. We discuss these techniques below and the impact of our result on them.

5.1 System-Call-Based Intrusion Detection Techniques

Many host-based Instruction Detection Systems (IDSs) monitor the behavior of an application process at the system call level. These systems build abstract models of a program based on system call traces. At runtime, the IDS monitors the system calls issued by the program. Any deviation from the pre-built abstract model is considered abnormal or incorrect behavior of a program. One of the earliest attempts was by Forrest et al. [18][23] in which short sequences of system calls (*N-grams*) obtained from training data are used to define a process's correct behavior. The monitoring is a matter of sequence matching against the pre-built *N-gram* database. Wagner and Dean [48] build abstract system call models from the control flow graph based on static source code analysis. A basic Non-Deterministic Finite Automaton (NDFA) and a more powerful Non-Deterministic Pushdown Automaton (NPDA) that incorporates stack state are built. Sekar et al. [34] improve Forrest's training method by building a Finite State Automaton (FSA) constructed from training system traces by associating system calls with program counter information. Feng et al. [19] further improve the training method in the VtPath model. At every system call, VtPath extracts the virtual stack list, which is the list of return addresses of functions in the call stack. Then a virtual path is extracted from two consecutive virtual stack lists and stored as a string in a hash table. VtPath detects some attacks that are missed by the FSA model. In a follow-up paper, Feng et al. [17] propose a static version of VtPath, called VPStatic, and compare it to DYCK by Griffin et al. [21], which constructs PDA models directly from binary code. Gao et al. [22] propose the execution graph model that uses training system call traces to approximate the control flow graph that is usually only available through static code analysis. The execution graph is built by considering both program counter and call stack information.

All these intrusion detection methods monitor process behavior at the system-call level, that is, they are only triggered upon system calls. As shown in this paper, non-control-data attacks require no invocation of system calls, therefore the attacks will most likely evade detection by system-call based monitoring mechanisms. Data flow information needs to be incorporated in these IDS models in order to detect non-control-data attacks.

Some IDS techniques [25] abstract a program's normal behavior using statistical distributions of system call parameters. The distribution is obtained from training data. At runtime, the IDS detects program anomalies by observing deviations from the training model. These methods detect intrusions based on the anomalies of data rather than the anomalies in control flow. Therefore, we believe that. with proper training, they can detect some of the attacks presented in this paper, in particular, the HTTPD CGI-BIN attack when */bin/sh* is run by the *execve(),* since that is most likely not in the training model. The method, however, is not able to detect the decision-making data attack described in Section 4.4, where no system call parameter is modified. Nor can it detect the user-identity data attack discussed in Section 4.1 without considering control flow information in the training model. It might be difficult for a statistical algorithm to precisely extract a fine-grained policy to detect the attacks with a reasonably low false positive rate. Despite such technical difficulties, considering system call parameter anomalies is one possible way to extend current IDSs to detect some non-control-data attacks.

5.2 Control Data Protection Techniques

Corrupting control data to alter the control flow is a critical step in traditional attacks. Compiler techniques and processor-architecture-level techniques have been proposed in very recent papers to protect control data. DIRA is a compiler to automatically insert code only to check the integrity of control data [35]. An explicitly stated justification for this technique is that control-data attacks are currently considered the most dominant attacks. Suh, Lee, and Devadas develop the *Secure Program Execution* technique to defeat memory corruption attacks [42]. The idea is to tag the data directly or indirectly derived from I/O as *spurious data*, a concept more commonly referred to as *tainted data* in other literature [16][30][44]. Security attacks are detected when tainted data is used as an instruction or jump target addresses. Another recent work on control data protection is *Minos* [10], which extends each memory word with an *integrity* bit. *Integrity* indicates whether the data originating from a trusted source. It is essentially the negation of taintedness. Very similar to *Secure Program Execution*, *Minos* detects attacks when the integrity bit of a control data is 0.

We agree that control data are highly critical in security-related applications. Not protecting them allows attacks to succeed easily. However, the general applicability of non-control-data attacks suggests the necessity of improvements of these techniques for better security coverage.

5.3 Non-Executable-Memory-Based Protections

A number of defensive techniques are based on non-executable-memory pages, which block an attacker's attempt to inject malicious code onto a writable memory page and later divert program control to execute the injected code. *StackPatch* is a Linux patch to disallow executing code on the stack [41]. Microsoft has also implemented non-executable-memory page supports in Windows XP Service Pack 2 [1]. In addition, the latest versions of Linux and OpenBSD are enhanced with similar protections.

These defensive techniques cannot defeat non-control-data attacks because there is no attempt to run any injected code during these attacks. Note that non-executable-memory-based protections can also be defeated by the *return-to-library* attacks, which divert program control to library code instead of the injected code [33].

5.4 Memory Safety Enforcement

CCured [27] is a program transformation tool that attempts to statically verify that a C program is type-safe, and thus free from memory errors. When static analysis is insufficient to prove type-safety, it instruments vulnerable portions of code with checks to avoid errors such as NULL pointer dereferences, out-of-bounds memory accesses, and unsafe type casts. Its main mechanism for enforcing memory safety is a type-inference algorithm that distinguishes pointers by how safely they are used. Based on this classification of pointers, code transformations are applied to include appropriate runtime checks for each type of pointer. Although *CCured*'s analysis techniques and runtime system are sophisticated and guarantee memory safety, instrumented programs often incur significant performance overheads and require nontrivial source code changes to ensure compatibility with external libraries.

CRED [53] is a buffer overflow detector that uses the notion of referent objects to add bounds checking to C without restricting safe pointer. Any addresses resulting from arithmetic on a pointer must lie within the same memory object as that of the pointer. To enforce this, *CRED* stores the base address and size of all memory objects in the object table. Immediately before an in-bounds pointer is used in an arithmetic operation, its referent object's bounds data is retrieved from the object table. This data is used to ensure the pointer arithmetic's result lies within the bounds of the referent object. When an out-of-bounds address is used in pointer arithmetic, its associated referent object's bounds data is used to determine if the resulting address is in bounds. To reduce performance overhead, *CRED* limits its bounds checking to string buffers, which implies that *CRED* does not provide protection against attacks involving non-string buffers. In addition, programs that perform heavy string-processing (e.g., web/email servers) can still incur overheads as high as 200%.

Cyclone [24] is a memory-safe dialect of C that aims to maintain much of C's flexible, low-level nature. It ensures safety in C by imposing a number of restrictions. Like *CCured*, *Cyclone* adds several pointer types that indicate how a pointer is used and inserts appropriate runtime checks based on a pointer's type. Porting C programs to *Cyclone*, however, can be difficult due to its additional restrictions and semantics.

For example, *Cyclone* only infers pointer kinds for strings and arrays. As such, it is often the programmer's responsibility to determine the appropriate type for a pointer. This task can be very time-consuming for large programs that make extensive use of pointers. In addition, *Cyclone* programs often perform significantly worse than their C counterparts and commonly-used software development tools such as compilers and debuggers must be modified for use with *Cyclone* source code.

Although the techniques enforcing memory-safety continue to show great promise, the software engineering community has not established techniques that allow an easy migration path from current large code bases. Moreover, the high overheads that they incur make them unsuitable for many kinds of software, particularly highly trafficked servers. Due to these reasons, memory-safety bugs are likely to still exist for an extended period of time. Hence, research efforts should still be invested in defensive techniques that assume the existence of memory-safety bugs.

5.5 Other Defensive Techniques

We now discuss other runtime defensive techniques that do not assume the control-data attack pattern.

Specialized Techniques Not Affected by Non-Control-Data Attacks. The effectiveness of some specialized dynamic detection techniques is not affected by non-control-data attacks. *StackGuard* [14] and *Libsafe* [7] can still defeat many stack buffer overflow attacks unless security sensitive data are in the same frame as the overflowing buffer, as in the GHTTPD example. *FormatGuard* [8] is still effective to defeat format string attacks because it does not allow overwriting of arbitrary memory addresses. However, these techniques are not generic enough to defeat attacks exploiting other types of vulnerabilities.

Generic Techniques Requiring Improvement. Among the various techniques that address a broader range of memory vulnerabilities, the underlying principles of the pointer protection technique *PointGuard* [9], address-space randomization techniques [4][6][51], and *TaintCheck* [28] are sound, but improvements are needed to better deploy these principles, as follows:

PointGuard is a compiler technique that embeds pointer encryption/decryption code to protect pointer integrity in order to defeat most memory corruption attacks. In principle, if all pointers, including pointers in application code and in libraries (e.g., LibC), are encrypted, most memory corruption attacks can be defeated. However, without the instrumented library code, the current *PointGuard* cannot defeat many non-control-data attacks. For example, the previously presented heap overflow and format string attacks only corrupt heap free-chunk pointers and the argument pointers of *printf*-like functions, which are pointers in LibC. Although there are technical challenges in the instrumentation of *PointGuard* at the library level (e.g., the lack of accurate type information), we argue that such an improvement is essential.

The principle of address-space randomization techniques is to rearrange memory layout so that the actual addresses of program data are different in each execution of the program. Ideally, the addresses should be completely unpredictable. Nevertheless, Shacham et al. [43] have recently shown that most current randomization implementations on 32-bit architectures suffer from the low entropy problem: even with very aggressive re-randomization measures, these techniques cannot provide more than 16-20 bits of entropy, which is not sufficient to defeat determined intruders. Deploying address-space randomization techniques on 64-bit machines is considered more secure.

TaintCheck [28] uses a software emulator to track the taintedness of application data. Depending on its configuration and policies, *TaintCheck* can perform checks on a variety of program behaviors, e.g., use of tainted data as jump targets, use of tainted data as format strings, and use of tainted data as system call arguments. Preventing the use of tainted data as system call arguments can be used to detect some, but not all of the attacks described in this paper. However, as the authors of *TaintCheck* have pointed out, this can lead to false positives, as some applications require legitimately embedding tainted data in the system call arguments. Further, the reported runtime slowdown is between 5-37 times. Further research is required to address the issues of security coverage, false positive rate, and performance overhead.

5.6 Defeating Memory Corruption Attacks: A Challenging Problem in Practice

The above analysis shows that finding a generic and practical technique to defeat memory corruption attacks is still an challenging open problem. The specialized techniques can only defeat attacks exploiting a subset of memory vulnerabilities. As for generic defensive techniques, many of them provide security by enforcing control flow integrity, and thus the security coverage is incomplete due to the general applicability of non-control-data attacks. A few other generic solutions, although not fundamentally relying on control flow integrity, need improvements to overcome the practical constraints on their deployment.

6 Empirical Discussions of Mitigating Factors

Despite the general applicability of non-control-data attacks, which is the main thesis of this paper, we experienced more difficulty in constructing these attacks than in constructing control-data attacks. In particular, the requirement of application-specific semantic knowledge and problems presented by the lifetime of security-critical data are major mitigating factors that impose difficulties on attackers.

6.1 Requirement of Application-Specific Semantic Knowledge

An obvious constraint for constructing non-control-data attacks is the attacker's reliance on application-specific knowledge. In a control-data attack, as long as a function pointer or a return address can be overwritten, a generic piece of shell code will be started to do all kinds of security damage easily. However, a non-control-data attack must preserve control flow integrity, so an attacker needs to have in-depth knowledge of how the target application behaves. For example, to attack HTTP servers, we need insights into the CGI mechanism; to attack the WU-FTP server, we should know how the effective UID is elevated and dropped. At the current stage, we have not formulated an automatic method to obtain such knowledge. The method we used in attack construction is a combination of vulnerability report review, debugger-aided source code review. and certain diagnostic tools such as *strace* (system call tracer) and *ltrace* (library call tracer). This is a time-consuming process. As a result, we have not succeeded in the attempt to attack *Sendmail* through an integer overflow vulnerability [40], as we are not as familiar with *Sendmail* semantics as we are with HTTP, SSH, FTP, and Telnet.

However, we argue that this is not a fundamental constraint on attackers for two reasons: 1) Knowledge of widely used applications is not hard to obtain, so a determined attacker is likely to eventually succeed no matter how long it takes, if there is a strong incentive; 2) Although we spent a great deal of effort to construct these attacks, future attackers may not need to expend the same amount of effort. For example, if a new vulnerability is found in another HTTP server, an attacker could easily think of attacking the CGI-BIN configuration. This can be thought of as similar to the history of the stack-smashing attack; it was a mystery when the Morris Worm spread, but it is now straightforward to understand.

6.2 Lifetime of Security-Critical Data

Lifetime of security-critical data is another constraint on seeking non-control-data attacks. The lifetime of a value is defined as the interval between the time the value is stored in a variable and the time of its last reference before the being de-allocated or reassigned. Only when a vulnerability is exploitable during the lifetime of some security-critical data can an attack succeed.

Our experience shows that although there are many potential data critical to security, a majority of them are eliminated by the constraint of data lifetime – the vulnerability occurs either before the data value is initialized or after the semantically diverging operation is performed. Therefore, reducing the lifetime should be considered as a secure programming practice. Two of the discussed attacks would not succeed if the programs were slightly changed as shown in Table 5. The original WU-FTPD function *getdatasock()* uses the global data *pw->pw_uid* in the *seteuid* call, allowing any vulnerability occurring before *getdatasock()* to escalate the process privilege. If the function was written as (A2), where a short-living local variable is used, only a vulnerability occurring within the lifetime of *tmp* (denoted as a bidirectional arrow) could affect the *seteuid* call. Similarly, in the original SSHD *do_authentication()* (code B1), the lifetime of the *authenticated* value covers the vulnerable *packet_read()* call. By inserting the statement "*authenticated=0*" after Line L1 in code B2, *authenticated* flag is always refreshed in every iteration, and thus its lifetime becomes shorter. The attack could not succeed since the vulnerability in L1 was out of the lifetime of *authenticated* flag.

The lifetimes of security-critical configuration data, as those in the *NULL HTTPD* attack and the *Telnetd* attack, are more difficult to reduce. A possible protection solution is to encrypt them in a way similar to the encryption technique used by *PointGuard* or to set the memory of configuration data read-only.

Table 5: Reducing Data Lifetime for Security

```
 (A1) Original WU-FTPD getdatasock()
 { seteuid(0);
   setsockopt( ... );
▼  seteuid(pw->pw_uid);
 }


 (A2) Modified WU-FTPD getdatasock()
▲{ tmp = geteuid();
   seteuid(0);
   setsockopt( ... );
▼  seteuid(tmp);
 }
```

```
 (B1)Original SSHD do_authentication()
▲{ int authenticated = 0;
   while (!authenticated) {
 L1:type = packet_read(); //vulnerable
    switch (type) {
      case SSH_CMSG_AUTH_PASSWORD:
       if (auth_password(user, passwd))
            authenticated = 1;
     case ...
    }
▼   if (authenticated) break;
   }
   do_authenticated(pw);
 }

 (B2)Modified SSHD do_authentication()
 { int authenticated = 0;
   while (!authenticated) {
 L1:type = packet_read(); //vulnerable
 ▲  authenticated = 0;
    switch (type) {
      case SSH_CMSG_AUTH_PASSWORD:
       if (auth_password(user, passwd))
            authenticated = 1;
     case ...
    }
 ▼  if (authenticated) break;
   }
   do_authenticated(pw);
 }
```

7 Related Work

Our research is motivated by a number of papers investigating system susceptibilities under hardware transient errors. It has been shown that random hardware faults can lead to security compromises in many real-world applications. Boneh et al. [5] show that the Chinese Remainder Theorem based implementation of the RSA signature algorithm is vulnerable to any hardware/software errors during certain phases of the algorithm. The produced erroneous cipher text allows the attacker to derive the RSA private key. In [50], we observe that even single-bit flip transient errors in critical sections of server programs can cause false authentications. We also show in an experiment [15] that bit-flip errors in Linux kernel firewall facilities allow malicious packets to survive firewall packet filtering. Govindavajhala and Appel conduct a real physical fault injection experiment with a spotlight bulb heating the PC memory chips [20]. The Java language type system can be subverted with high probability under this harsh condition. Although the results in the context of random errors may not demonstrate imminent security threats, they clearly indicate the possibility of finding attacks other than altering control flow in real-world systems.

Also related are papers discussing the possibility of evading system-call-based host IDS's by disguising traces of system calls. Mimicry attacks [48][49] cannot be detected by the IDS because the malicious code can issue system call sequences that are considered legitimate under the IDS model. The attacks proposed by Tan et al. evade IDS detection by changing foreign system calls to equivalent system calls used by the original program [45]. It should be noted that mimicry attacks still alter program control flow, and thus are defeated by control-flow-integrity-based protections and non-executable-memory-based protections.

Pincus and Baker conduct a study on memory vulnerabilities and attacks [29]. They extract three primitive attack techniques and provide a taxonomy of current defensive techniques. A conclusion of the study is that current defensive techniques are not comprehensive; each one only provides partial coverage, and no combination of them defeats all known attacks.

8 Conclusions

We begin with the Applicability Claim: *many real-world software applications are susceptible to attacks that do not hijack program control flow, and the severity of the resulting security compromises is equivalent to that of control-data attacks.* The claim is empirically validated by experiments constructing non-control-data attacks against many major network server applications. Each attack exploits a different type of memory vulnerability to corrupt non-control data and obtain the privilege of the victim process. Based on the results of the experiments, we argue that control flow integrity may not be a sufficiently accurate approximation of software security. The general applicability of non-control-data attacks represents a realistic threat to be considered seriously in defense research.

We study a wide range of current defensive techniques and discuss how the general applicability of non-control-data attacks affects the effectiveness of these techniques. The analysis shows the necessity of further research on defenses against memory corruption based attacks. Finding a generic and secure way to defeat memory corruption attacks is still an open problem.

Despite their general applicability, non-control-data attacks are less straightforward to construct than are control-data attacks, because the former require semantic knowledge about target applications. Another important constraint is the lifetime of security-critical data. We suggest that reducing data lifetime is a secure programming practice that increases software resilience to attacks.

Acknowledgments

We owe thanks to many people for their insightful suggestions and extensively detailed comments on the technical contents and the presentation of this paper. In particular, we thank Peng Ning at North Carolina State University, Fei Chen, John Dunagan, Jon Pincus, Dan Simon, and Helen Wang at Microsoft, and Fran Baker, Zbigniew Kalbarczyk, and Karthik Pattabiraman at University of Illinois at Urbana-Champaign. The comments from the anonymous reviewers have also improved the paper.

This work is supported in part by a grant from Motorola Inc. as part of Motorola Center for Communications, in part by NSF ACI CNS-0406351, and in part by MURI Grant N00014-01-1-0576.

References

[1] S. Andersen and V. Abella. Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies. http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx

[2] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(7), Nov. 1996.

[3] The Apache Software Foundation. http://www.apache.org/

[4] PaX Address Space Layout Randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt

[5] D. Boneh, R. A DeMillo, and R. Lipton. On the importance of eliminating errors in cryptographic computations. In *Proceedings of Advances in Cryptology: Eurocrypt '97*, pp.37-51, 1997

[6] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of 12th USENIX Security Symposium.* Washington, DC, August 2003.

[7] A. Baratloo, T. Tsai, and N. Singh, Transparent run-rime defense against stack smashing attacks, In *Proceedings of USENIX Annual Technical Conference*, June 2000.

[8] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. FormatGuard: Automatic protection from printf format string vulnerabilities. *In Proceedings of the 10th USENIX Security Symposium*, Washington, DC, August 2001.

[9] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium.* Washington, DC, August 2003.

[10] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. To appear in *Proceedings of the 37th International Symposium on Microarchitecture*. Portland, OR. December 2004.

[11] CERT Security Advisories. http://www.cert.org/advisories/

[12] CERT CC. CERT Advisory CA-2001-33 Multiple Vulnerabilities in WU-FTPD, 2001.

[13] CERT Advisory CA-2001-21 Buffer Overflow in telnetd. http://www.cert.org/advisories/CA-2001-21.html

[14] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Automatic detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998.

[15] S. Chen, J. Xu, R. K. Iyer, and K. Whisnant. Modeling and analyzing the security threat of firewall data corruption caused by instruction transient errors, In *Proceedings of* the *IEEE International Conf. on Dependable Systems and Networks*, Washington DC, June 2002.

[16] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, Jan/Feb 2002

[17] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, May 2004.

[18] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longsta. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, May 1996.

[19] H. Feng, O. Kolesnikov, P. Fogla, W. Lee and W. Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, May 2003.

[20] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *Proceedings* of *IEEE Symposium on Security and Privacy,* 2003, Oakland, California. May 2003.

[21] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *Proceedinsg of the*

Symposium on Network and Distributed System Security, February 2004.

[22] D. Gao, M. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In *Proceedings of the 11th ACM Conference on Computer and Communication Security*, October 2004.

[23] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3), 1998.

[24] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of USENIX Annual Technical Conference*. Monterey, CA, June 2002.

[25] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *Proceeding of ESORICS 2003*, October 2003.

[26] Microsoft Security Bulletin, http://www.microsoft.com/technet/security/

[27] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2002

[28] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS '05)*, February 2005.

[29] J. Pincus and B. Baker. Mitigations for Low-level Coding Vulnerabilities: Incomparability and Limitations. http://research.microsoft.com/users/jpincus/mitigations.pdf, 2004.

[30] Perl Security. http://www.perldoc.com/perl5.6/pod/perlsec.html

[31] K. Pekka and L. Kalle. SSH1 Remote Root Exploit. http://www.hut.fi/~kalyytik/hacker/ssh-crc32-exploit_Korpinen_Lyytikainen.html. 2002

[32] P. Starzetz. CRC32 SSHD Vulnerability Analysis. http://packetstormsecurity.org/0102-exploits/ssh1.crc32. txt

[33] R. Wojtczuk. Defeating Solar Designer Non-executable Stack Patch. http://geek-girl.com/bugtraq, January 1998.

[34] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001.

[35] A. Smirnov and T. Chiueh. DIRA: Automatic detection, identification and repair of control-data attacks. In *Proceedings of the 12th Network and Distributed System Security Symposium (NDSS),* San Diego, CA, February 3-4, 2005.

[36] Ghttpd Log() Function Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/5960

[37] SSH CRC-32 Compensation Attack Detector Vulnerability. http://www.securityfocus.com/bid/2347/

[38] Null HTTPd Remote Heap Overflow Vulnerability. http://www.securityfocus.com/bid/5774 and http://www.securityfocus.com/bid/6255

[39] Multiple Vendor Telnetd Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/3064

[40] Sendmail Debugger Arbitrary Code Execution Vulnerability. http://www.securityfocus.com/bid/3163

[41] Solar Designer. StackPatch. http://www.openwall.com/linux

[42] G. Suh, J. Lee, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. Boston, MA. October 2004.

[43] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address space randomization. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. Washington, DC. Oct. 2004.

[44] U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[45] K.M.C. Tan, K.S. Killourhy and R.A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. *RAID*, 2002.

[46] Tim Newsham. Format String Attacks. http://muse.linuxmafia.org/lost+found/format-string-attacks.pdf

[47] United States Computer Emergency Readiness Team. Technical Cyber Security Alerts, http://www.us-cert.gov/cas/techalerts/

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

[49] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, 2002.

[50] J. Xu, S. Chen, Z. Kalbarczyk, R. K. Iyer, An experimental study of security vulnerabilities caused by errors. In *Proceedings of the IEEE International Conf. on Dependable Systems and Networks*, Göteborg, Sweden, July 01-04, 2001.

[51] J. Xu, Z. Kalbarczyk and R. K. Iyer. Transparent Runtime Randomization for Security. In Proceedings of Symposium on Reliable and Distributed Systems (SRDS), Florence, Italy, October 6-8, 2003.

[52] W. Young and J. McHugh. Coding for a believable specification to implementation mapping, In *Proceedings of the IEEE Symposium on Security and Privacy*, 1987.

[53] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, February 2004.