View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace@MIT

DETECTING BUFFER OVERFLOWS USING TESTCASE SYNTHESIS AND CODE INSTRUMENTATION

by

MICHAEL A. ZHIVICH

S.B., Electrical Engineering and Computer Science (2004)
Massachusetts Institute of Technology

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2005

© Massachusetts Institute of Technology 2005. All rights reserved.

Author .....
Department of Electrical Engineering and Computer Science
May 19, 2005

Certified by .....
Richard Lippmann
Senior Scientist, MIT Lincoln Laboratory
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Detecting Buffer Overflows Using Testcase Synthesis and Code Instrumentation

by

Michael A. Zhivich

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The research presented in this thesis aims to improve existing approaches to dynamic buffer overflow detection by developing a system that utilizes code instrumentation and adaptive test case synthesis to find buffer overflows and corresponding failure-inducing inputs automatically. An evaluation of seven modern dynamic buffer overflow detection tools determined that C Range Error Detector (CRED) is capable of providing fine-grained buffer access information necessary for the creation of this system. CRED was also selected because of its ability to provide comprehensive error reports and compile complex programs with reasonable performance overhead. CRED was extended to provide appropriate code instrumentation for the adaptive testing system, which also includes a test case synthesizer that uses data perturbation techniques on legal inputs to produce new test cases, and an analytical module that evaluates the effectiveness of these test cases. Using information provided by code instrumentation in further test case generation creates a feedback loop that enables a focused exploration of the input space and faster buffer overflow detection. Applying the adaptive testing system to `jabberd`, a Jabber Instant Messaging server, demonstrates its effectiveness in finding buffer overflows and its advantages over existing dynamic testing systems. Adaptive test case synthesis using CRED to provide buffer access information for feedback discovered 6 buffer overflows in `jabberd` using only 53 messages, while dynamic testing using random messages generated from a protocol description found only 4 overflows after sending 10,000 messages.

Thesis Supervisor: Richard Lippmann
Title: Senior Scientist, MIT Lincoln Laboratory

Acknowledgments

I am indebted to many people without whom this work would not be possible. First and foremost, I would like to thank my advisor, Richard Lippmann, for his guidance, suggestions and constructive criticism throughout this project. I would also like to thank Tim Leek for all his contributions to this work, including the real exploit evaluation of dynamic buffer overflow detection tools and modifications to the PCFG random message generator and CRED-HW compiler. Without Tim's contributions and technical expertise, building an adaptive testing system would have been very difficult. I would like to thank Misha Zitser for developing a thorough classification of buffer overflows and providing the test corpus from his thesis, which was an invaluable part of the evaluation of dynamic buffer overflow detection tools. I would also like to thank Rob Cunningham, Graham Baker, Kendra Kratkiewicz, Chris Scott and all the folks in Group 62 at the Lincoln Laboratory, with whom I have had many interesting discussions, and who have made my experience at Lincoln so rewarding.

Last, but certainly not least, I would like to thank my family and friends for their encouragement and support throughout this time — I would not have made it here without you, and I give you my heartfelt thanks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The use of computers and software has become essentially ubiquitous in modern society. Making a call on a mobile phone, using an ATM, providing electricity — almost everything in modern life directly or indirectly is supported by computers running software. Such broad reliance on software requires it to be secure and reliable — if it were not, the results could be catastrophic. While failures experienced by consumer electronics, such as a mobile phone rebooting because of a fault, are hardly disastrous, a failure in a mission-critical system can become life-threatening very quickly. For example, if control software in a nuclear power plant crashes or sends the wrong control signals, an explosion or reactor meltdown could occur and cause serious casualties in addition to lasting environmental damage. While such scenarios may seem far-fetched, other incidents, such as online fraud and identity theft, are affecting people every day because vulnerabilities in software enable crackers to obtain this information. Thus, finding and fixing vulnerabilities in software is necessary to avert potential disasters and utilize the full potential of available technology.

1.1 Cost of Software Failures

While significant resources are already invested to test software and find mistakes, errors still exist in deployed code and result in very serious, if not catastrophic situations. A buffer overflow was partly responsible for the blackout experienced by the Northeast in August of 2003. Due to a software error that was caused by abnormal conditions, the supervisory system failed to update the data on displays and notify operators of a serious strain on the physical infrastructure that triggered a collapse. The fault lay dormant in the software for

a number of years, and was only revealed under abnormal circumstances [19]. This service outage affected 50,000,000 people in the US and Canada and caused damage estimated at 7–10 billions of dollars [11, 42]. While not a security failure, this incident demonstrates that vulnerabilities exist even in mission-critical software.

Another major outage of a public service occurred in January of 1990 within the telephone network owned by AT&T. A failure of a single node triggered an error in the failure-recovery routine of 114 other nodes that provided long distance service, thus bringing the entire network down. The culprit was an incorrectly used `break` command in an `if` block nested inside a `switch` statement. The resulting outage lasted nine hours and affected an estimated 60,000 people. AT&T estimated lost revenue to be approximately 60 million dollars; however, this estimate does not include lost revenue of businesses that relied on the telephone network and were affected by the outage [5].

Many recent software faults have been exploited via cyber-attacks. In particular, popular targets are buffer overflows that exist in software providing web services. Most recently, worms such as Blaster and CodeRed overloaded and brought down corporate networks by rapidly infecting a large number of computers. The spread of the infection generated so much traffic that some networks were rendered completely unusable. Blaster exploited a buffer overflow in Microsoft's RPC service, while CodeRed exploited a buffer overflow in IIS indexing service [6, 7]. The cost of lost productivity and necessary repairs due to Blaster has been estimated at 300–500 million dollars, and the estimated cost of CodeRed exceeds 1.2 billion dollars [33, 53].

Exploiting faults in software results in huge financial losses for businesses, loss of privacy for consumers, and substantial amount of time and effort wasted by both trying to repair the damage. Currently, the only method of preventing exploitation of these faults is patching them as soon as the vulnerability is known and software developers can fix the problem. Unfortunately, patching is not an efficient or practical method of addressing this issue. For some companies, especially those whose business centers around e-commerce (like Amazon.com or Ebay), taking servers offline to perform updates is very expensive, and doing so every month is impractical. In addition, only faults that are discovered by well-meaning security experts get patched quickly. Faults discovered by malicious hackers may be exploited for a long time before the exploits are noticed and software developers are alerted to the problem.

1.2 Types of Vulnerabilities in Software

In order to discuss different types of software vulnerabilities, it is useful to establish some terminology. A software *fault* or *error* is a mistake that a programmer has made while creating the program or an unintended consequence of interaction between several programs running together. A *vulnerability* is a fault that can be exploited to make a program perform actions that its creator has not intended. An *exploit* is an input or sequence of actions that uses a vulnerability to obtain higher privileges on a computer system, execute arbitrary code or perform some other action that the program being attacked was not intended to do under these circumstances.

The previous section described several incidents relating to faults in software. While many different kinds of software faults can lead to devastating consequences, not all software faults are security vulnerabilities. Many different types of errors can occur in a program, including buffer overflows, race conditions, dangling pointers, integer overflows and underflows, divide by zero errors, truncation errors, failures to reduce privileges, time of check to time of use errors, etc. The list is seemingly endless. While it is difficult for a software developer to ensure that none of these bugs creep into the program, it is even harder to test for these conditions automatically. Some faults, such as the one that caused the August 2003 blackout, occur under very obscure conditions and are triggered by input sequences that are very unlikely. Since testing all possible inputs is frequently impractical, such inputs are likely not explored during testing. Some faults, such as race conditions, are very hard to replicate, and may often appear as Heisenbugs[1].

Certain kinds of faults can be exploited by a malicious person to take over the computer system and inflict damage on a usually unsuspecting victim. Although many different kinds of faults can be exploited, buffer overflows appear to be a recurring target for attacks on today's Internet. Attacks are also becoming more complicated, using buffer overflows as stepping stones to obtaining higher privileges and taking control of machines.

[1]Heisenbugs are named after Heisenberg's Uncertainty Principle. A Heisenbug disappears or alters its behavior when an attempt is made to prove or isolate it [58].

1.3 Buffer Overflows — A Persistent Vulnerability

Today's server software is under constant scrutiny and attack, whether for fun or for profit. Figure 1-1 shows the cumulative number of exploits found in commonly used server software, such as `IIS`, `BIND`, `Apache`, `sendmail`, and `wu-ftpd`. The data comes from the NIST ICAT database [37], and includes exploits across all versions of the software. Exploits are plotted according to the date they have been recorded in the ICAT database. The arrows indicate appearances of major worms, such as `Lion`, `CodeRed` and `Welchia`. As the data demonstrates, new vulnerabilities are still found, even in code that has been used and tested for years. A recent analysis by Rescorla agrees with this observation, as it shows that vulnerabilities continue to be discovered at a constant rate in many types of software [48].

Buffer overflows enable a large fraction of exploits targeted at today's software. Such exploits range from arbitrary code execution on the victim's computer to denial of service (DoS) attacks. During 2004, NIST recorded 889 vulnerabilities in its ICAT database, of which 75% are remotely exploitable. NIST further reports that 21% of the remotely exploitable vulnerabilities are due to buffer overflows. Over the past three years, the fraction of remotely exploitable vulnerabilities ranged from 75% to 80%, and buffer overflows constituted 21%–22% of these vulnerabilities [38]. These statistics show that buffer overflows comprise a significant fraction of existing vulnerabilities and that new buffer overflow vulnerabilities are still being discovered. Detecting and eliminating buffer overflows would thus make existing software far more secure.

1.4 Approaches to Buffer Overflow Prevention

The most desirable solution is, of course, to write perfect software. Since humans are imperfect, the next best solution is to find and eliminate faults in software before it is deployed. Even as stated, the problem is very difficult and, arguably, undecidable. However, extensive testing and validating of software before deployment decreases the chance that a fault will be found in production and require patching of a running system.

Available solutions that try to address the problem of software faults apply at different stages of software development. Some approaches try to standardize and regulate how the software is written, what language and libraries are used, etc. Such policy approaches help create better software by formalizing the development process, requiring code reviews

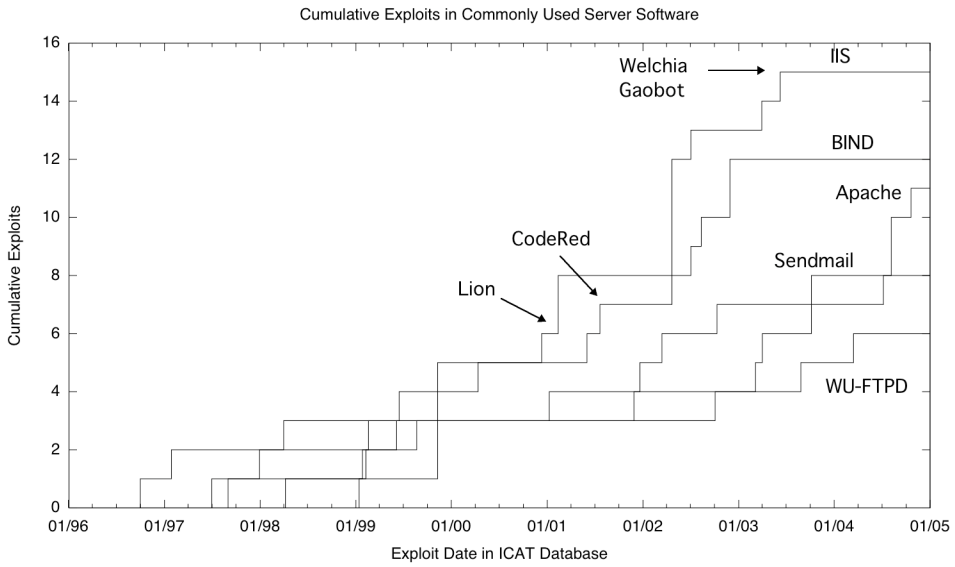

Figure 1-1: Cumulative exploits in commonly used server software. *Exploits across all versions of server software are plotted against the date they were recorded in the ICAT database [37]. Arrows indicate appearances of major worms.*

and organizing programmers' efforts. However, such policies are often difficult to enforce due to deadlines, insufficient training or lack of proper incentives for the developers. This approach does not protect from all errors, as programmers performing the code reviews are not perfect either; however, it is a useful and necessary component to creating robust and secure software.

The opposite extreme of the available approaches are ones that try to foil exploits at runtime. Few are successful at creating an environment where all exploits can be stopped, since many exploits rely on actions that the program is technically allowed to take (though usually under different circumstances). Such tools range from compiler-based instrumentation to executable monitoring and operating system extensions. All such approaches come with performance overhead; furthermore, more comprehensive protection results in heavier additional computational overhead. Thus, sophisticated protection of this sort is rarely used in production systems, since performance, not security, is more critical to company revenues.

An attractive and more realistic approach exists in-between. While developers will unavoidably create imperfect software, many of the mistakes can be found and fixed in the testing stage, before the product is deployed and vulnerabilities can be exploited. NIST

reports that the annual cost to the US economy due to software errors is estimated at 60 billion dollars; however, the cost could be reduced by over a third through improved software testing [36]. Furthermore, the cost of fixing a defect once software has been released in production is estimated to be up to 15 times more than the cost of the corresponding fix during the testing phase [57]. While unit-testing and creating integration tests is a responsibility that software developers must assume, it is generally very difficult and time-consuming. In addition, schedules and deadlines usually leave little time for testing, and time allocated for Quality Assurance (QA) is often cut first when the project is behind schedule.

An attractive testing approach is an automated system that can analyze source code, synthesize test cases and then run tests on the program. While test cases verifying functionality have to be created by developers, testing *robustness* can, to some extent, be performed automatically. Such tests include verifying program's ability to handle malformed inputs and its response to large volume of inputs. Several products have been developed that statically analyze software source to find patterns that correspond to faults. Even better analyzers can verify, in some cases, that accesses to a particular buffer do not exceed the upper bound, regardless of the input. However, these analyses are very computation-intensive, and the tools frequently do not scale to large programs, where it is much harder to find faults.

An alternative to statically analyzing software is dynamic testing. The source code can be instrumented at the time of compilation and then tested on automatically-generated test cases. There are thus two challenges to the dynamic testing approach — finding the right granularity of instrumentation for the source code and generating inputs that trigger the latent faults. Current state of the art dynamic testing systems, such as the one developed in the PROTOS project at University of Oulu [28], use randomly-generated test cases based on an attribute grammar that describes the protocol of program inputs. The PROTOS system has been particularly effective in finding buffer overflows in open source server software, which is widely deployed on many systems comprising today's Internet.

1.5 Goals of this Thesis

The goal of this thesis is to improve on the state of the art in dynamic testing and create a system that offers faster and more comprehensive buffer overflow detection. To achieve this, methods for instrumenting executables and automatically synthesizing test cases are investigated. Specifically, this thesis aims to do the following:

- Evaluate capabilities of modern dynamic buffer overflow detection tools and select a tool to provide code instrumentation for subsequent system development,
- Develop a system that uses code instrumentation to improve the random message testing approach developed by the PROTOS project,
- Design and build an adaptive testing system that uses code instrumentation and feedback in test case generation to enable more efficient and comprehensive buffer overflow detection.

Chapter 2

Exploiting Buffer Overflows

Before discussing the tools and approaches available for buffer overflow detection, it is useful to understand attacks that have been used to exploit vulnerabilities stemming from buffer overflows. This chapter provides some background information about memory layout of processes and offers a brief overview of different types of buffer overflow attacks.

2.1 Memory Layout of a Process

In order to understand how buffer overflow exploits work, it is important to know the memory layout of a process. The particular details described in this chapter pertain to Unix/Linux processes, and while some details may be different, the memory layout of processes on Windows is not fundamentally different.

Figure 2-1 presents a diagram of different memory regions in a running program. The lower memory addresses are shown on the bottom while the higher addresses are on the top. The memory for a process is divided into five different segments — *text*, *data*, *bss*, *heap* and *stack*. The *text* segment contains the assembly instructions that are executed by the processor. This memory segment is usually mapped *read-only*, as the program should not need to modify its own instructions. The *data* segment contains global and static variables that have been initialized to some non-zero value in the program code. The *bss* segment contains all other, uninitialized, global and static variables. Both of these segments are mapped *read-write* into the memory space of a process.

The *heap* memory region is grown dynamically to accommodate requests of the program. Memory is requested via calls to `malloc`, `realloc` or `calloc` and then released by using `free`

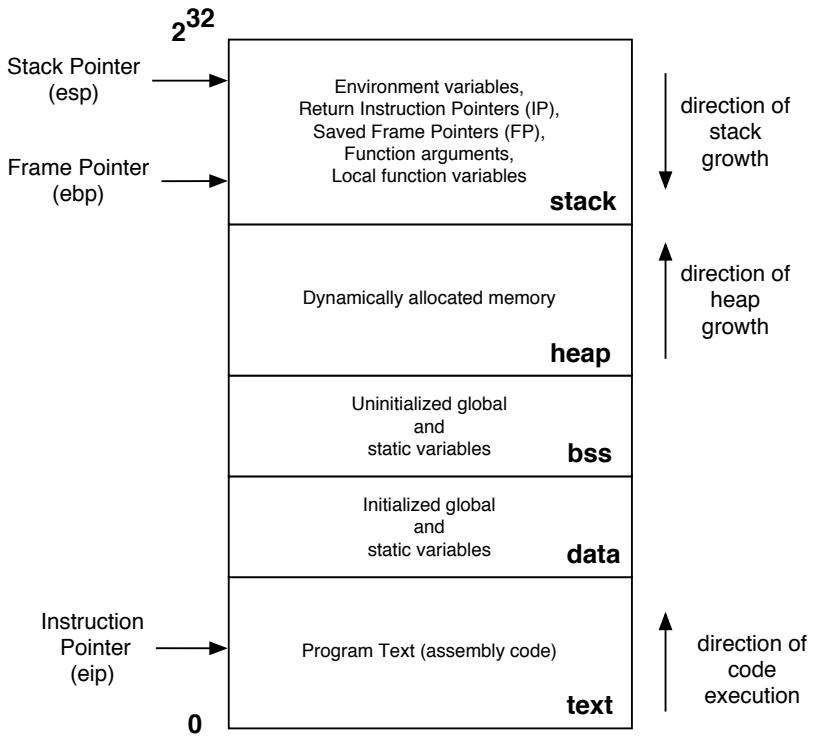

Figure 2-1: Memory layout of a process. *This diagram shows different memory segments of a process. The text segment is mapped read-only into the process memory space, while all other segments are mapped read-write. The instruction, stack and frame pointers include the names of x86 registers that store these values.*

when the memory is no longer needed. As shown, the heap memory region starts in lower address space and expands up as necessary. The *stack* segment is used for environment variables, function arguments, local variables and keeping record of the execution path. Whenever a function is called, a new *stack frame* is pushed onto the stack, containing the address of the instruction to which the process should return once the execution of the function is completed, as well as the value of the *frame pointer* that points to beginning of the stack frame currently in use. As shown, the stack starts at a high memory address (`0xC0000000` for Linux systems) and expands downwards as necessary. Both heap and stack memory regions are mapped *read-write* into the memory space of a process.

The diagram also shows the typical contents of three registers that affect program execution. The *instruction pointer* (the `eip` register on x86) determines which instruction is executed. In general, it contains some address in the *text* segment of the program, though some Linux signal handlers have been known to put code in the *stack* segment and then

redirect the `eip` to point to the *stack* segment. The *stack pointer* (the `esp` register on x86) determines the next free memory location on the stack. It always points to the *stack* segment. The *frame pointer* (`ebp` register on x86) determines the location of the current stack frame. It always points to the *stack* segment as well. While not strictly necessary, the frame pointer is frequently used in calculating addresses of local variables on the stack. Since `ebp` always points to the location of the saved `ebp` from the previous stack frame, frame pointers can be used to determine the call chain that is stored on the stack.

2.2 Arbitrary Code Execution

Arbitrary code execution is one of the most severe consequences of an exploit, since the attacker can execute any code (usually a shell) with the privileges of the process that has been exploited. Frequently, such an attack is used as a stepping stone in obtaining higher privileges, since most remotely accessible processes, such as webservers, do not run as root. However, there are many exploits available to increase the privilege level from local user to root, thus giving the attacker full control of the compromised computer. Thus, the attacker can read and change sensitive files, modify permissions and install backdoors.

Two steps are needed to achieve this result, and there are multiple methods of completing each of these steps. First, the attacker needs to inject the code to be executed (usually assembly code for the system call `execve("/bin/sh")`). Then, he needs to change a code pointer so that it points to the injected code and then arrange for the code pointer to be used.

2.2.1 Injecting Code

The code is typically injected when the program copies some outside input into a buffer. Thus, the attack code could be residing in a request packet from the network, in an environment variable, or in a carefully-crafted file. The code is typically very short (only 46 bytes are needed to execute a shell), and thus fits into even a reasonably small buffer. For simple attacks, such as the "stack-smashing" attack [1], the attack code is inserted into the buffer that is being overflowed to change the code pointer. However, this is not necessary — attack code can be inserted into any buffer in the program, as long as the attacker knows the memory address of the beginning of the attack code. This address is then inserted into a changed code pointer, as described below.

2.2.2 Changing Code Pointer Directly

A variety of code pointers are targeted by attackers in order to change the execution path to contain injected code. Frequent targets include the return instruction pointer (IP) that is saved on the stack and function pointers that are frequently used by programs that need a way to describe handlers for different events. A direct way of changing code pointers is via a *continuous overflow* of a buffer located in the same memory region as the target code pointer. A *continuous overflow* is defined as an access pattern that consists of accesses to several consecutive elements in the buffer, including one or more bytes outside the boundary of the buffer. Such overflows frequently result from using string or I/O functions, such as `gets` or `strcpy`, as these functions copy consecutive bytes from the input stream or another buffer (respectively) into the target buffer. If the input is too long or the source buffer is larger than the destination buffer, then the destination buffer will be overflowed — the program will write memory that has not been allocated for use as part of this buffer.

As discussed above, the values for return instruction pointers are saved in the stack segment of process memory. Thus, stack buffers are often targeted, since an overflow in a stack buffer would enable an attacker to change the return IP to a desired value. Function pointers can be stored on the stack, on the heap or in the data or bss segments. An overflow in a buffer in the appropriate segment is needed to overwrite a function pointer. In addition, buffers are filled "up" — from low memory address to high memory address. Thus, in order to overwrite a return IP or a function pointer, an attacker needs to find a buffer that is stored lower in memory than the target of the attack.

2.2.3 Changing Code Pointer Indirectly

It is not necessary to find a buffer that can be overflowed continuously until the target code pointer is reached. Frequently, it is sufficient to find an overflow that allows the attacker to change some pointer in the program. For example, a 4-byte overflow is sufficient to change a pointer that is stored above a stack buffer in memory to contain the address of the return IP on the stack. If the attacker can also control the value written to memory when the overwritten pointer is used, then the return IP can be changed indirectly, without a continuous overflow.

Sometimes it is possible to use a single write to change a code pointer. Buffers are frequently aliased and accessed through pointers by using offsets. If this offset is miscalculated, the write can access memory outside of the buffer bounds. Often, such offset is controlled by some property of the input, such as length of a token or position of some delimiter in the input. The attacker can thus control the value of the offset and overwrite the code pointer with the desired value. This is an example of a *discrete overflow* — a single access outside the buffer bounds that is calculated via an offset from an in-bounds pointer.

Another target of discrete overflows is the saved frame pointer. A fake stack frame can be created by an attacker elsewhere, and the saved frame pointer changed to point to this stack frame. If the fake stack frame is created on the stack, its location is likely very close to that of the real stack frame, and thus only a single byte of the saved frame pointer needs to change to point it to the fake stack frame. This can be accomplished with a discrete overflow. Upon return from the current function, the program will use the fake stack frame that contains a return IP pointing to injected code, thus giving the attacker control of the program when the next return instruction is executed.

Another frequent attack target is a `longjump` buffer. This buffer is used to change program flow without using the stack. A `setjump` command sets up such a buffer, containing a saved frame pointer and a return IP, much like a stack frame. A `longjump` command can then use this buffer to change the execution path and jump to the location in the program specified by such buffer. It is a convenient target as an attacker can modify the values of return IP or saved frame pointer to hijack the process. Such buffers are frequently stored on the stack, and the necessary values could be changed by either a discrete or a continuous overflow.

2.3 Logic-based Attacks

Another class of attacks focuses on changing program flow by altering variables that control the logic of the program. While such attacks usually do not result in arbitrary code execution, they frequently allow an attacker to raise his privilege level or perform some other action the program is not intended to do. For example, if a buffer with an exploitable overflow is located close to a flag that specifies access permissions, the attacker can overflow the buffer and change his permissions. Alternatively, suppose that a program writes to a file,

whose filename is stored next to a buffer that an attacker can overflow. If the program is running as `root`, then by changing the filename to `‘‘/etc/passwd’’` and providing appropriate input the attacker can create an account for himself on the target system. A famous logic-based attack was used in the Morris worm, that exploited `sendmail` by turning on its debug mode, thus making it execute commands on the worm's behalf [51]. Logic attacks are possible in any memory region that can contain a buffer, and while they are not as common as some other attacks, they are equally dangerous.

2.4 Unauthorized Access to Information

Many attacks are designed to give an attacker control or higher privilege level on the target system, and they require an existence of an exploitable overflow that allows the attacker to write something to memory. However, *read overflows* can also be exploited to gain unauthorized access to information. Suppose that `printf` is used to print a buffer that has not been properly null-terminated. Then, `printf` will continue printing from memory until it encounters a `NULL` byte. If such an exploitable buffer is stored next to a cryptographic key, for example, the attacker will have gained very sensitive information. Since many string and I/O functions in the standard C library do not ensure null-termination of buffers, this exploit may sometimes occur with an appropriate input.

For performance reasons, programmers generally do not clear memory before they release it, so many memory locations in the heap (and stack) regions still store contents of previously used buffers. If a buffer with an exploitable read overflow exists nearby, an attacker can use it to gain potentially sensitive information. Such an attack was recently discovered in Google's popular Gmail service — if an e-mail containing a malformed address was sent, the program displayed a large chunk of memory that was stored nearby, frequently including other people's e-mails, which could have contained confidential information [43].

2.5 Classifying Buffer Overflows

As discussed in previous sections, different classes of attacks can be used to exploit buffer overflows. The characteristics of a buffer overflow determine which attack can be leveraged against it, and how successful such an attack will be. This section summarizes some of these properties in an attempt to classify different kinds of buffer overflows. A more thorough

taxonomy of buffer overflows with corresponding examples can be found in [62]. The properties of buffer overflows discussed here will prove important in constructing test cases for the evaluation of dynamic buffer overflow detection tools, which is presented in Chapter 4.

2.5.1 Access Pattern

The *access pattern* describes whether a buffer overflow is *continuous* or *discrete*. A *continuous* overflow results in modification or exposure of several consecutive bytes outside the buffer boundary, and is achieved by consecutively accessing buffer elements and running past the upper bound. Such overflows are present in "stack-smashing" attacks, and attacks that utilize string functions. A *discrete* overflow usually results in a single access to memory outside the buffer bounds, due to a miscalculation of an offset from an in-bounds pointer. This kind of overflow can be used in logic-based attacks or attacks that modify a pointer, since changing a single byte of a pointer can be sufficient to exploit it.

2.5.2 Access Type

The *access type* describes whether a buffer overflow is a *read* or a *write* overflow. While *write* overflows give an attacker more control over the program that is executing, *read* overflows are also dangerous, as they can compromise sensitive information. Read overflows are also much more difficult to detect — since write overflows modify memory, an attacker must be very careful, as any mistake can cause program state to become corrupted, crashing the program and thus exposing the attack. Many read overflows will go completely undetected, as they do not corrupt program state.

2.5.3 Size of Overflow

The size of an overflow is likewise important in determining potential damage. Unfortunately, even off-by-one overflows can be exploitable, so miscalculating the buffer size by one byte may lead to serious consequences. Small overflows are much harder to detect than large overflows, since program state corruption may not be obvious — the goal of a small overflow may be to overwrite some internal variable resulting in a logic-based attack. Very large read and write overflows are easily detectable, as they will generate a segmentation fault — eventually the overflow will try to access memory beyond the boundary of the current memory page, and the operating system will report this error. However, the size of

a memory page on a modern Linux system is 4096 bytes, so a very large overflow is needed to trigger this behavior.

2.5.4 Location

Buffers can exist in four different memory regions — *stack*, *heap*, *data* and *bss*. As discussed above, different memory regions serve to separate memory segments designed for different purposes. The buffers in the *stack* memory segment are more likely targets for attacks, since overwriting return IP or function pointers on the stack is very attractive. However, depending on the location of the target code pointer or variable controlling program flow, a buffer in any of the above memory locations can potentially be used in an exploit.

Chapter 3

Approaches to Buffer Overflow Detection

Various methods for finding and preventing buffer overflows have been developed, including policy techniques, such as enforcing secure coding practices and mandating the use of safe languages, halting exploits via operating system extensions, statically analyzing source code to find potential overflows and detecting buffer overflows at runtime [13]. Each approach has its advantages; however, each also suffers from limitations. This chapter discusses some of the available approaches and highlights their strengths and weaknesses. It also presents motivation for an adaptive testing approach that improves the dynamic testing method by mitigating some of its limitations.

3.1 Security Policies and Code Reviews

Due to a large number of security vulnerabilities in software many companies have devoted extensive resources to ensuring security. Considerable time and effort is expended on implementing and enforcing secure coding procedures, following proper software development cycle and conducting code reviews. Microsoft has recently announced its renewed focus on security and claims to have performed a large-scale review of its Windows codebase as part of its Trustworthy Computing initiative [31]. This code review involved up to 8,500 developers and lasted several months [45]. However, bugs remain in Windows code, as evidenced by security patches for Windows Server 2003, released after the security review [30]. This anecdote demonstrates that it is very difficult to verify by inspection that a complex piece

of software contains no faults, and thus code reviews, no matter how thorough, will miss bugs. While policies and procedures encouraging secure coding and proper development practices aid in developing secure software, they cannot replace testing or guarantee the security of resulting code.

3.2 Language Approach

An alternative that guarantees absence of buffer overflows is writing code in a "safe" language. Languages such as Java provide built-in bounds checking for arrays and buffers, and thus programs written in Java are immune to buffer overflows, forgoing errors in JVM implementation. However, large legacy codebases have been written in C; furthermore, many developers are opposed to switching to a different language, commonly citing worse performance as the reason. Efforts have been made to create languages based on C that provide similar capabilities. CCured [35] and Cyclone [25] are good examples of such languages.

CCured works by executing a source to source transformation of C code and then compiling the resulting code with gcc. CCured performs a static analysis to determine pointer types (`SAFE`, `SEQ` or `WILD`). Instrumentation based on pointer type is inserted into the resulting source code whenever CCured cannot statically verify that the pointer will always be within the bounds of the buffer. This instrumentation changes pointer representation, so code compiled with CCured cannot interoperate easily with code compiled with gcc — wrapper functions are needed to add or remove metadata used by the instrumentation whenever an object passes between instrumented and uninstrumented code. Since CCured compiler cannot handle large programs, the developer may be required to re-write or annotate 2-3% of the code in order to enable CCured to perform the necessary transformations. At runtime, the resulting executable performs bounds-checking and terminates program with an error message if a buffer access violation has been detected.

Cyclone is a dialect of C that has been developed to provide bounds-checking for C-like programs. Like CCured, it uses a combination of static and dynamic analysis to determine pointer types and only instruments pointers that require it. In order to make proper bounds information available at runtime, Cyclone changes C's representation of pointers to include additional information, such as bounds within which a pointer is legal. A major disadvantage of Cyclone is that existing C code needs to be ported to the Cyclone language before it

can be compiled with the Cyclone compiler. In order to convert the code, a developer is required to rewrite about 10% of the code, inserting special Cyclone commands. Furthermore, Cyclone differs enough from C that some C programmers may be reluctant to use it [25].

The main disadvantage of such safe languages is that they tend to be slower than C, and provide less control to the programmer. Another concern is the difficulty of translating the large amount of legacy code still in use today that has been written in C into a safe language. A good example is `sendmail`, in which multiple buffer overflows have been found over the years [37]. The effort to translate such programs into a safe language would be immense and the cost unacceptable. Some programs, such as CCured, aim for an automatic source to source transformation of C into a safe language; however, they generally require programmer intervention and fail to run successfully on large programs.

3.3 Safe C Libraries

A large class of overflows resulting in exploits is enabled by unsafe C library functions. Many functions in the C library are written to provide fast performance and relegate the responsibility of calculating appropriate buffer bounds to the programmer. Frequently, programmers miscalculate these bounds or else do not realize that the library function can overflow the buffer. Several alternative libraries, that claim to provide a "safe" implementation of functions in the C library have been developed; some are discussed here. Libsafe [2, 3, 54] is a dynamically loaded library that intercepts calls to functions in glibc that are considered dangerous. These routines include `strcpy, strcat, getwd, gets, scanf, realpath, sprintf` and others. The call to one of these routines is replaced with a wrapper function that attempts to compute the maximum size for each local buffer. The size of a buffer is computed as the distance from the start of the buffer to the saved frame pointer above it. If the size limitation allows it, the intercepted function is executed normally; otherwise, the program is terminated, thus turning a buffer overflow into a denial of service attack. Although Libsafe protects against a stack-smashing attack, it does not prevent overwriting of local variables, which can be used to change a code pointer indirectly. This method also fails if the executable has been compiled with `-fomit-frame-pointer` option, as the frame pointer is no longer saved on the stack. This option is frequently enabled during compilation to increase performance in the resulting executable.

Libverify [3] is an enhancement of Libsafe. During each function call, the return address is copied onto a special "canary stack" stored on the heap. When the function returns, the return address on the stack is compared to the address on the "canary stack" — if the two match, the program is allowed to proceed; otherwise, the program is terminated and an alert is issued. Libverify protects against a larger class of attacks than Libsafe; however, attacks using function pointers still remain undetected and lack of protection for the "canary stack" on the heap is a major limitation.

3.4 Operating System Extensions

Various methods for thwarting exploits based on buffer overflows using operating system extensions have been proposed and deployed. A commonly-used tactic is runtime randomization of stack location and C library entry points [8]. This method does not prevent the overflow from occurring; however, it effectively converts the attack into denial of service. Since attacks that overwrite code pointers need to use an absolute address, randomizing stack location and C library entry points makes it very difficult for the attacker to guess the correct address value to change a code pointer. Thus, the guess will likely be incorrect, and the program will segfault when it tries to execute some random memory as instructions. The advantage of this method is that it is very lightweight and completely transparent to the program being executed; however, buffer overflows are not prevented, so any attack that does not rely on changing code pointers will still succeed.

3.5 Static Analysis

Static analysis seems like an attractive approach, since little additional effort from the programmer is needed to analyze code — the code is examined automatically and no test cases are required. Misha Zitser evaluated the capabilities and effectiveness of five modern static analysis tools, including both commercial (PolySpace C Verifier) and open source (ARCHER, BOON, SPLINT, UNO) products [52, 18, 22, 56, 60]. The tools were tested using historic versions of `wu-ftp`, `bind` and `sendmail` that contained real-world vulnerabilities [62]. The study found that none of the tools could process the entire source code of these programs; thus, model programs containing the buffer overflow vulnerabilities were extracted from the source of the full programs.

Two tests were performed on the resulting model programs, which were several hundred lines in size. Each static analysis tool was first executed on the model program containing the vulnerability, and a *detection* was recorded if the tool correctly identified the vulnerability. A patch was then applied to fix the vulnerability, such that no buffer overflow existed in the model program. Each static analysis tool was then executed on the resulting program, and a *false alarm* was recorded if the tool still indicated that the vulnerability existed.

The study found that most static analysis tools have a very low detection rate. PolySpace C Verifier and SPLINT had a higher detection rate, but also suffered from a high false alarm rate. Combined with the tools' inability to handle the entire source of the programs, these characteristics make static analysis an impractical method for detecting buffer overflows [62].

3.6 Runtime Buffer Overflow Detection

A different approach to mitigating the problem of buffer overflows is to stop the attacks that exploit them at runtime. Such tools either use instrumentation inserted into the executable at compile time or monitor the binary executable itself to determine when an out-of-bounds access has occurred. By stopping the attack in its tracks, these tools convert the attack into a denial of service.

3.6.1 Compiler-based Code Instrumentation

Some of the runtime buffer overflow detection tools work by inserting appropriate instrumentation at compile time. The instrumentation depends on the particular approach and the attacks the tool is trying to prevent.

Stack-smashing Protection

StackGuard [12] and ProPolice [17] take a similar approach to protecting an executable against a stack-smashing attack. During compilation, the prologue and epilogue of each function call is modified so that a "canary" value is inserted between the stack frame, which contains saved return IP and frame pointer, and local variables. Upon return from a function, this value is checked to ensure that it has not changed; if it has, there must have been an overflow attack that could have damaged the return IP — thus, the program is terminated with an error. StackGuard provides more sophisticated options, including

a terminator canary (consisting of four bytes — `NULL`, `'\r'`, `'\n'`, `EOF`). Such a canary provides better protection when the buffer is overflowed using a string library function, as these bytes serve as termination characters. By default, a random value picked at compile time is used for the "canary."

A major limitation of this approach is that only continuous overflows that overwrite the canary on the stack are detected. Many other indirect attacks can still change the value of the return IP. ProPolice tries to mitigate this problem by rearranging local variables such that character stack buffers are directly below the canary value. While this method helps prevent some indirect stack attacks, it does not protect any heap buffers. Read overflows and underflows also remain undetected.

Fine-grained Bounds Checking

TinyCC [4] and CRED [50] both provide fine-grained bounds checking based on the "referent object" approach, developed by Jones and Kelly [26]. As buffers are created in the program, they are added to an *object tree*. For each pointer operation, the instrumentation first finds the object to which the pointer currently refers in the object tree. The operation is considered illegal if it references memory or results in a pointer outside said object. When an illegal operation is detected, the program is halted. TinyCC is a small and fast C compiler that uses a re-implementation of Jones and Kelly code; however, it is much more limited than gcc and provides no error messages upon an access violation — the program simply segfaults. CRED, on the other hand, is built upon Jones and Kelly code, which is a patch to gcc. Thus, CRED is able to compile almost all programs that gcc can handle. In addition, CRED adheres to a less strict interpretation of the C standard and allows illegal pointers to be used in pointer arithmetic and conditionals, while making sure that they are not dereferenced. This treatment of out-of-bounds pointers significantly increases the number of real world programs that can be compiled with this instrumentation, as many programs use out-of-bounds pointers in testing for termination conditions.

An interesting approach to mitigating buffer overflows via fine-grained bounds checking has been developed by Martin Rinard, as part of *failure-oblivious computing* [49]. Instead of halting program execution when an out-of-bounds access has been detected, a program compiled with the failure-oblivious compiler will instead ignore the access and proceed as if nothing has happened. This approach prevents damage to program state that occurs due

to out-of-bounds writes and thus keeps the program from crashing or performing actions it is not intended to do. Instrumentation likewise detects out-of-bounds reads and generates some value to return — for example, returning `NULL` will stop read overflows originating in string functions. The paper discusses several other values that may drive the program back into a correct execution path. This compiler is implemented as an extension to CRED and thus also uses "referent object" approach.

Insure++ [44] is a commercial product that also provides fine-grained bounds checking capabilities. The product is closed-source so little is known about its internal workings. According to its manual, Insure examines source code and inserts instrumentation to check for memory corruption, memory leaks, memory allocation errors and pointer errors, among other things. The resulting code is executed and errors are reported when they occur.

The main limitation of all these tools is that an overflow within a structure is not recognized until the border of the structure has been reached. CRED is capable of detecting overflows within structures correctly, but only when structure members are accessed directly — if an access is made through an alias pointer, CRED will detect the overflow only when it reaches the border of the structure. These tools also cannot detect overflows within uninstrumented library functions. These issues are common to all compiler-based approaches, and are discussed further in Section 4.1.3. TinyCC seems to have some incorrect wrappers for library functions and cannot handle large programs, which is a significant drawback. Another limitation of TinyCC is that no error messages are provided — the program simply terminates with a segmentation fault. Insure's major flaw is the performance slowdown that it imposes on executables — some programs run up to 250 times slower.

3.6.2 Executable Monitoring

Another method of detecting and preventing buffer overflows at runtime is through executable monitoring. Chaperon [44] is an executable monitor that is part of the Insure toolset from Parasoft, and Valgrind [27] is an open-source project aimed at helping developers find buffer overflows and prevent memory leaks. Tools such as Chaperon and Valgrind wrap the executable directly and monitor calls to `malloc` and `free`, thus building a map of heap memory in use by the program. Buffer accesses can then be checked for validity using this memory map.

There are several limitations to this approach. Since no functions are explicitly called to allocate and deallocate stack memory, these tools can monitor memory accesses on the stack only at a very coarse-grained level, and will not detect many attacks exploiting stack buffers. Another limitation is the large performance overhead imposed by these tools. Since Valgrind simulates program execution on a virtual x86 processor, testing is 25–50 times slower than gcc.

Program shepherding [29] is an approach that determines all entry and exit points of each function, and ensures that only those entry and exit points are used during execution. Thus, an attacker cannot inject code and jump to it by changing a code pointer — this will be noticed as an unspecified exit point and the program will be halted. Like operating system approaches discussed in Section 3.4, this method is transparent to the program running on the system. However, it is much more heavy-weight and incurs higher performance overhead. In addition, attacks that do not rely on changing code pointers will still succeed, as they do not change entry and exit points of a function.

A similar approach is used by StackShield [55]. One of the security models supported by this tool is a Global Return Stack. Return IP for each function that is entered is stored on a separate stack, and the return pointer is compared with the stored value when the function call is completed. If the values match, the execution proceeds as intended; however, if the values do not match, the value on the return stack is used as the return address. Thus, the program is guided back to its intended path. This approach detects attacks that try to overwrite the return IP or provide a fake stack frame. However, this method does not stop logic-based attacks or attacks based on overwriting a function pointer.

Another security model offered by StackShield is Return Change Check. Under this model a single global variable is used to store the return IP of the currently executing function, and upon exit the return IP saved on the stack is compared to the one saved in this variable. If they do not match, the program is halted and an error message is displayed. This security model also tries to protect function pointers by making the assumption that all function pointers should point into the *text* segment of the program. If the program tries to follow a function pointer that points to some other memory segment, it is halted and an alert is raised. While this method does not prevent logic-based attacks from proceeding, it makes attacking the system more difficult. Another limitation of this approach is that read overflows are not detected.

3.7 Software Fault Injection

A popular approach for testing program robustness is a dynamic analysis technique focused on injecting faults into software execution. This technique creates anomalous situations by injecting faults at program interfaces. These faults may force a particular program module to stop responding or start providing invalid or malformed data to other modules. By observing system behavior under such anomalous conditions, certain inherent vulnerabilities can be discovered. Tools such as FIST [21] and Fuzz [32] have been used to perform such testing, with some promising results.

The main drawback of software fault injection is that it requires considerable developer intervention and effort. The code must be manually prepared for testing and the developer needs to analyze the output of each test case reporting a system crash to determine where the error has occurred. Automating program execution with fault injection is also difficult, as the program is likely to crash or hang due to inserted faults.

3.8 Mini-simulation with Random Messages

An interesting dynamic testing approach has been developed at the University of Oulu, Finland. As part of the PROTOS project [28], researchers developed a mini-simulation system for functional testing. PROTOS tests applications that use standard protocols, such as HTTP, TCP, SNMP, etc; however, if a protocol grammar is provided, PROTOS can test an arbitrary application. Source code for the application is not necessary, as PROTOS works by testing executables directly. An attribute grammar is used to describe the protocol, and this description is used by the tester to automatically generate a large number of anomalous and normal test cases. The application under test is then executed on each input, and abnormal behavior, such as crashing or hanging is reported. Making the tests fully automatic is difficult, as the application being tested may need to be restarted. PROTOS has been used to test different applications including HTTP browsers, an LDAP server and SNMP agents and management stations. Of the 49 different product versions that were tested, 40 were found vulnerable to at least a denial-of-service attack.

While mini-simulation presents an automatic approach for finding buffer overflows, the method used to detect overflows is very coarse. Many small overflows will not cause a segfault and will thus be missed by the tool; however, these can still be exploited to alter

program logic and change code pointers, as discussed in Chapter 2. Most read overflows are not detected either, as they do not trigger a segfault unless they access memory outside a valid page. In addition, no information is provided to the developer about where the error occurred.

Another disadvantage of using the mini-simulation approach is that a *revealing input* that triggers the overflow is required; however, the input space for a program is generally very large. While the attribute grammar provides the mini-simulation system with a way to generate malformed inputs, it has no knowledge of how to generate revealing inputs. Thus, the input space is explored randomly and focusing on particular code paths is not possible.

3.9 An Adaptive Testing System

The mini-simulation system described above appears to be the most comprehensive tool available today for testing program robustness. In order to improve on this approach, the remainder of this thesis presents research aimed at mitigating the limitations of the mini-simulation approach. Some of the limitations can be addressed by instrumenting source code for the application to provide fine-grained bounds-checking capabilities. An instrumented executable will detect more buffer overflows and provide the developer with information necessary to locate the overflow. Chapter 4 presents an evaluation of modern dynamic buffer overflow detection tools, which could be used for this purpose. Chapter 5 describes a modified mini-simulation system that makes use of an instrumented executable in testing programs with random messages.

A partial solution to the problem of finding revealing inputs is a feedback loop that will allow the tester to make use of information about buffer access patterns in the executable. If such information were available, the tester could correlate the mutations in the input with variations in the buffer access pattern, and thus evaluate the effectiveness of a particular mutation. Then, the search through the input space can be directed by observing the changes in the access pattern and generating inputs that will bring the buffer closer to overflow. This method is explored as part of the adaptive test case synthesis, presented in Chapter 6.

Chapter 4

Evaluating Dynamic Buffer Overflow Detection Tools

This chapter is a minor extension of a paper entitled "Dynamic Buffer Overflow Detection" by Michael Zhivich, Tim Leek and Richard Lippmann [61]. This paper has been submitted to the 2005 Symposium on the Foundations of Software Engineering. Section 4.4 describing Tim Leek's evaluation of dynamic buffer overflow detection tools on model programs appears in this paper, and has been included here to maintain continuity.

In order to develop an approach that builds on the state of the art in dynamic testing, it is necessary to have the ability to detect buffer overflows of all sizes and in all memory segments. The resulting system should be able to handle large, complex programs and support developer-defined extensions. This chapter focuses on evaluating the effectiveness of current dynamic buffer overflow detection tools with the above goals in mind. A similar evaluation has been conducted by Wilander et al. [59], but it focused on a limited number of artificial exploits which only targeted buffers on the stack and in the bss section of the program. Our evaluation reviews a wider range of tools and approaches to dynamic buffer overflow detection and contains a more comprehensive test corpus, including Wilander's artificial exploits.

The test corpus consists of three different testsuites. Section 4.2 presents the results for *variable-overflow* testsuite, which consists of 55 small test cases with variable amounts overflow, specifically designed to test each tool's ability to detect small and large overflows in different memory regions. Section 4.3 describes the results for *artificial exploit* testsuite, which consists of 18 artificial exploits from Wilander's evaluation [59] that utilize buffer overflows to produce a shell. Section 4.4 presents the results for 14 model programs containing remotely exploitable buffer overflows extracted from `bind`, `wu-ftpd` and `sendmail`.

The chapter is organized as follows: Section 4.1 presents an overview of the tools tested in this evaluation, Sections 4.2 – 4.4 present a description and results for three different test-suites, Section 4.5 describes performance overhead incurred by the tools in this evaluation, and Section 4.6 summarizes and discusses our findings.

4.1 Dynamic Buffer Overflow Detection Tools

This evaluation tests modern runtime buffer overflow detection tools including those that insert instrumentation at compile-time and others that wrap the binary executable directly. This section presents a short description of each tool, focusing on its strengths and weaknesses.

A summary of tool characteristics is presented in Table 4.1. A tool is considered to include *fine-grained bounds checking* if it can detect small (off-by-one) overflows. A tool *compiles large programs* if it can be used as a drop-in replacement for gcc and no changes to source code are needed to build the executable; however, minimal changes to the makefile are acceptable. The *error report time* specifies whether the error report is generated when the error occurs or when the program terminates. Since program state is likely to become corrupted during an overflow, continuing execution after the first error may result in incorrect errors being reported. Instrumentation may also be corrupted, causing failures in error checking and reporting. If a tool can protect the program state by intercepting out-of-bounds writes before they happen and discarding them, reporting errors at termination may provide a more complete error summary.

4.1.1 Executable Monitoring Tools

Chaperon [44] is part of the commercial Insure toolset from Parasoft. Chaperon works directly with binary executables and thus can be used when source code is not available. It intercepts calls to `malloc` and `free` and checks heap accesses for validity. It also detects memory leaks and *read-before-write* errors.

One limitation of Chaperon is that fine-grained bounds checking is provided only for heap buffers. Monitoring of buffers on the stack is very coarse. Some overflows are reported incorrectly because instrumentation can become corrupted by overflows. Like all products in the Insure toolset, it is closed-source which makes extensions difficult.

Tool	Version	OS	Requires Source	Open Source	Fine-grained Bounds Checking	Compiles Large Programs	Error Report Time
Wait for segfault	N/A	Any	No	Yes	No	Yes	Termination
GCC	3.3.2	Linux	No	Yes	No	Yes	Termination
Chaperon	2.0	Linux	No	No	No*	Yes	Occurrence
Valgrind	2.0.0	Linux	No	Yes	No*	Yes	Termination
CCured	1.2.1	Linux	Yes	Yes	Yes	No	Occurrence
CRED	3.3.2	Linux	Yes	Yes	Yes	Yes	Occurrence
Insure++	6.1.3	Linux	Yes	No	Yes	Yes	Occurrence
ProPolice	2.9.5	Linux	Yes	Yes	No	Yes	Termination
TinyCC	0.9.20	Linux	Yes	Yes	Yes	No	Termination

Table 4.1: Summary of tool characteristics. *(* = fine-grained bounds checking on heap only)*

Valgrind [27] is an open-source alternative to Chaperon. It simulates code execution on a virtual x86 processor, and like Chaperon, intercepts calls to `malloc` and `free` that allow for fine-grained buffer overflow detection on the heap. After the program in simulation crashes, the error is reported and the simulator exits gracefully.

Like Chaperon, Valgrind suffers from coarse stack monitoring. Also, testing is very slow (25 – 50 times slower than gcc [27]), since the execution is simulated on a virtual processor.

4.1.2 Compiler-based Tools

CCured [35] works by performing static analysis to determine the type of each pointer (`SAFE`, `SEQ`, or `WILD`). `SAFE` pointers can be dereferenced, but are not subject to pointer arithmetic or type casts. `SEQ` pointers can be used in pointer arithmetic, but cannot be cast to other pointer types, while `WILD` pointers can be used in a cast. Each pointer is instrumented to carry appropriate metadata at runtime - `SEQ` pointers include upper and lower bounds of the array they reference, and `WILD` pointers carry type tags. Appropriate checks are inserted into the executable based on pointer type. `SAFE` pointers are cheapest since they require only a `NULL` check, while `WILD` pointers are the most expensive, since they require type verification at runtime.

The main disadvantage of CCured is that the programmer may be required to annotate the code to help CCured determine pointer types in complex programs. Since CCured requires pointers to carry metadata, wrappers are needed to strip metadata from pointers when they pass to uninstrumented code and create metadata when pointers are received

from uninstrumented code. While wrappers for commonly-used C library functions are provided with CCured, the developer will have to create wrappers to interoperate with other uninstrumented code. These wrappers introduce another source of mistakes, as wrappers for `sscanf` and `fscanf` were incorrect in the version of CCured tested in this evaluation; however, they appear to be fixed in the currently-available version (v1.3.2).

C Range Error Detector (CRED) [50] has been developed by Ruwase and Lam, and builds on the Jones and Kelly "referent object" approach [26]. An *object tree*, containing the memory range occupied by all objects (i.e. arrays, structs and unions) in the program, is maintained during execution. When an object is created, it is added to the tree and when it is destroyed or goes out of scope, it is removed from the tree. All operations that involve pointers first locate the "referent object" – an object in the tree to which the pointer currently refers. A pointer operation is considered illegal if it results in a pointer or references memory outside said "referent object." CRED's major improvement is adhering to a more relaxed definition of the C standard – *out-of-bounds* pointers are allowed in pointer arithmetic. That is, an *out-of-bounds* pointer can be used in a comparison or to calculate and access an *in-bounds* address. This addition fixes several programs that generated false alarms when compiled with Jones and Kelly's compiler, as pointers are frequently tested against an *out-of-bounds* pointer to determine a termination condition. CRED does not change the representation of pointers, and thus instrumented code can interoperate with unchecked code.

The two main limitations of CRED are unchecked accesses within library functions and treatment of structs and arrays as single memory blocks. The former issue is partially mitigated through wrappers of C library functions. The latter is a fundamental issue with the C standard, as casting from a struct pointer to a `char` pointer is allowed. When type information is readily available at compile time, CRED detects overflows that overwrite other members within the struct; however, when a struct member is aliased through a type cast, the overflow remains undetected until the boundary of the structure is reached. These problems are common to all compiler-based tools, and are described further in Section 4.1.3.

Another gcc extension is **mudflap** [16], which is part of the gcc 4.0 release. Like CRED, mudflap inserts instrumentation at compile time by modifying the gcc abstract syntax tree to make calls to `libmudflap` whenever a pointer dereference occurs. At runtime, mudflap builds an *object tree* that includes all valid objects, such as arrays, structs and unions. This object tree is used during pointer dereference to determine whether the pointer is trying to

access memory that has been allocated. If no object exists at the memory location that the pointer is trying to access, the access is declared illegal and an alert is issued. Mudflap also protects a number of C library functions through wrappers that try to ensure that buffer bounds will not be overstepped.

Unlike CRED, mudflap does not use the "referent object" approach — that is, pointer arithmetic that transforms a pointer to one object into a pointer to another is allowed by mudflap. This limitation will result in mudflap allowing adjacent variables to be overwritten by an overflow. Some attempts are made to separate objects by inserting padding, so that continuous overflows between objects will be detected; however, this is not always possible — function arguments and structure members are assumed to be adjacent by uninstrumented code. Furthermore, discrete overflows that overwrite a valid memory area will also be missed. Although mudflap wraps a larger set of C library functions than CRED, the wrapper approach is in general an error-prone and incomplete solution. Since the source code for gcc 4.0 beta that included mudflap became available after this evaluation was completed, no formal evaluation of mudflap is included in this thesis.

Insure++ [44] is a commercial product from Parasoft and is closed-source, so we do not know about its internal workings. Insure++ examines source code and inserts instrumentation to check for memory corruption, memory leaks, memory allocation errors and pointer errors, among other things. The resulting code is executed and errors are reported when they occur.

Insure's major fault is its performance overhead, resulting in slowdown of up to 250 as compared to gcc. Like all tools, Insure's other limitation stems from the C standard, as it treats structs and arrays as single memory blocks. Since the product is closed-source, extensions are difficult.

ProPolice [17] is similar to StackGuard [12], and outperformed it on artificial exploits [59]. It works by inserting a "canary" value between the local variables and the stack frame whenever a function is called. It also inserts appropriate code to check that the "canary" is unaltered upon return from this function. The "canary" value is picked randomly at compile time, and extra care is taken to reorder local variables such that pointers are stored lower in memory than stack buffers.

The "canary" approach provides protection against the classic "stack smashing attack" [1]. It does not protect against overflows on the stack that consist of a single out-

of-bounds write at some offset from the buffer, or against overflows on the heap. Since ProPolice only notices the error when the "canary" has changed, it does not detect read overflows or underflows. The version of ProPolice tested during this evaluation protected only functions that contained a character buffer, thus leaving overflows in buffers of other types undetected; this problem has been fixed in currently-available versions by including `-fstack-protector-all` flag that forces a "canary" to be inserted for each function call.

Tiny C compiler (TinyCC) [4] is a small and fast C compiler developed by Fabrice Bellard. TinyCC works by inserting code to check buffer accesses at compile time; however, the representation of pointers is unchanged, so code compiled with TinyCC can interoperate with unchecked code compiled with gcc. Like CRED, TinyCC utilizes the "referent object" approach [26], but without CRED's improvements.

While TinyCC provides fine-grained bounds checking on buffer accesses, it is much more limited than gcc in its capabilities. It failed to compile large programs such as Apache with the default makefile. It also does not detect read overflows, and terminates with a segfault whenever an overflow is encountered, without providing an error report.

4.1.3 Common Limitations of Compiler-based Tools

There are two issues that appear in all of the compiler-based tools – unchecked accesses within library functions and treatment of structs and arrays as single memory blocks. The former problem is partially mitigated by creating wrappers for C library functions or completely reimplementing them. Creating these wrappers is error-prone, and many functions (such as File I/0) cannot be wrapped.

The latter problem is a fundamental issue with the C standard of addressing memory in arrays and structs. According to the C standard, a pointer to any object type can be cast to a pointer to any other object type. The result is defined by implementation, unless the original pointer is suitably aligned to use as a resultant pointer [46]. This allows the program to re-interpret the boundaries between struct members or array elements; thus, the only way to handle the situation correctly is to treat structs and arrays as single memory objects. Unfortunately, overflowing a buffer inside a struct can be exploited in a variety of attacks, as the same struct may contain a number of exploitable targets, such as a function pointer, a pointer to a `longjmp` buffer or a flag that controls some aspect of program flow.

4.2 Variable-overflow Testsuite Evaluation

The *variable-overflow* testsuite evaluation is the first in the series of three evaluations included in this chapter. This testsuite is a collection of 55 small C programs that contain buffer overflows and underflows, adapted from Misha Zitser's evaluation of static analysis tools [62]. Each test case contains either a *discrete* or a *continuous* overflow. A *discrete* buffer overflow is defined as an out-of-bounds write that results from a single buffer access, which may affect up to 8 bytes of memory, depending on buffer type. A *continuous* buffer overflow is defined as an overflow resulting from multiple consecutive writes, one or more of which is out-of-bounds. Such an overflow may affect an arbitrary amount of memory (up to 4096 bytes in this testsuite), depending on buffer type and length of overflow.

Each test case in the variable-overflow testsuite contains a 200-element buffer. The overflow amount is controlled at runtime via a command-line parameter and ranges from 0 to 4096 bytes. Many characteristics of buffer overflows vary. Buffers differ in type (`char`, `int`, `float`, `func *`, `char *`) and location (stack, heap, data, bss). Some are in containers (struct, array, union, array of structs) and elements are accessed in a variety of ways (index, pointer, function, array, linear and non-linear expression). Some test cases include runtime dependencies caused by file I/O and reading from environment variables. Several common C library functions (`(f)gets`, `(fs)scanf`, `fread`, `fwrite`, `sprintf`, `str(n)cpy`, `str(n)cat`, and `memcpy`) are also used in test cases.

4.2.1 Test Procedure

Each test case was compiled with each tool, when required, and then executed with overflows ranging from 0 to 4096 bytes. A 0-byte overflow is used to verify a lack of false alarms, while the others test the tool's ability to detect small and large overflows. The size of a memory page on the Linux system used for testing is 4096 bytes, so an overflow of this size ensures a read or write off the stack page, which should segfault if not caught properly. Whenever the test required it, an appropriately sized file, input stream or environment variable was provided by the testing script. There are three possible outcomes of a test. A *detection* signifies that the tool recognized the overflow and returned an error message. A *segfault* indicates an illegal read or write (or an overflow detection in TinyCC). Finally, a *miss* signifies that the program returned as if no overflow occurred.

Table 4.1 describes the versions of tools tested in our evaluation. All tests were performed on a Red Hat Linux release 9 (Shrike) system with dual 2.66GHz Xeon CPUs. The standard Red Hat Linux kernel was modified to ensure that the location of the stack with respect to *stacktop* address (`0xC0000000`) remained unchanged between executions. This modification was necessary to ensure consistent segfault behavior due to large overflows.

4.2.2 Sample Analysis

This section presents a sample analysis of results for a *discrete* and a *continuous* buffer overflow. Only results for gcc, ProPolice and CRED are discussed for these sample test cases. Each test case is compiled with the appropriate tool, and then executed for overflows ranging from 0 to 4096 bytes. The goal of this analysis is to demonstrate how the variable-overflow testsuite exposes weaknesses of different tools.

Discrete Overflow

For a discrete overflow, consider a test case shown in Figure 4-1 that includes a single out-of-bounds write at an adjustable index.

```
1  #define BUFF_SIZE 200
2
3  int main(int argc, char **argv) {
4      char buf[BUFF_SIZE];
5      int delta = atoi(argv[1]);
6
7      buf[BUFF_SIZE - 1 + delta] = 'a';
8
9      return 0;
10 }
```

Figure 4-1: Example of *discrete* overflow test case

Figure 4-3 presents the stack layout for this test case, as well as results for gcc, ProPolice and CRED. The "canary" is only present on the stack in executables compiled with ProPolice. Both gcc and CRED leave 16 bytes to save registers, which is not strictly necessary since `main` is a top-level function. ProPolice has a tighter stack layout and does not leave the space for saved registers. The graph shows the outcome of the test case for each amount of overflow, ranging from 1 to 4096 bytes. The outcome is one of *detection*, *miss* or *segfault*, as discussed in Section 4.2.1. For each integer amount of overflow in bytes, a *detection* is

indicated by an upward bar at the corresponding location, a *miss* is indicated by a hash mark on the line through the middle of the graph and a *segfault* is indicated by a downward bar.

Since compiling the program with gcc offers no protection, none of the overflows are detected. Overflows of 21–24 bytes correspond to overwriting the return instruction pointer (IP), as shown on the stack diagram. In most cases, the segfault behavior is caused by the program attempting to execute code at some invalid address, since one of the bytes of the return IP has been overwritten with 'a' (`0x61`). It is possible that changing one byte in the return IP results in a valid address and some unintended code is executed; however, the end result for all such test cases in this evaluation is a segfault. For large overflows, the write tries to access memory that is beyond the page boundary (above the stacktop address of `0xC0000000`), which also causes a segfault.

ProPolice does not perform much better. It detects the first four overflows, since the program is modifying the "canary" value. However, for discrete overflows larger than 4 bytes, the "canary" is unchanged, and the overflow remains undetected. Overflows of 5–8 bytes correspond to overwriting the saved frame pointer (FP), and overflows of 9–12 bytes correspond to overwriting the return IP, as shown on the stack diagram. Both are exploitable, and neither is detected. For large overflows, the behavior of the executable compiled with ProPolice is identical to that of one compiled with gcc.

CRED performs much better due to fine-grained bounds checking and "referent object" approach. The address for each write is checked with respect to the intended object, so all overflows are detected and the program is terminated before the write takes place. Since the address of the write is checked before executing it, there is no segfault behavior for very large overflows.

Continuous Overflow

A slightly more complicated test case is presented in Figure 4-4. The target buffer is enclosed in a struct, and a continuous overflow occurs inside the `strcpy` function.

Figure 4-6 presents the stack layout for this test case, as well as results for gcc, ProPolice and CRED. As with the discrete test case, the "canary" is only present in the ProPolice executable and 16 bytes for saved registers only appear in executables compiled with gcc and CRED. The graph likewise shows the outcome of the test case for each amount of

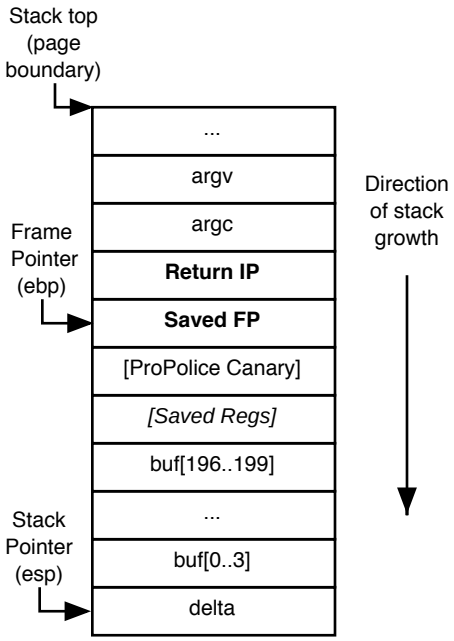

Figure 4-2: Stack layout for *discrete* buffer overflow example. *The example consists of a single out-of-bounds write (see Figure 4-1 for source code). ProPolice "canary" is present only in the ProPolice test case, and saved registers appear only in GCC and CRED test cases.*

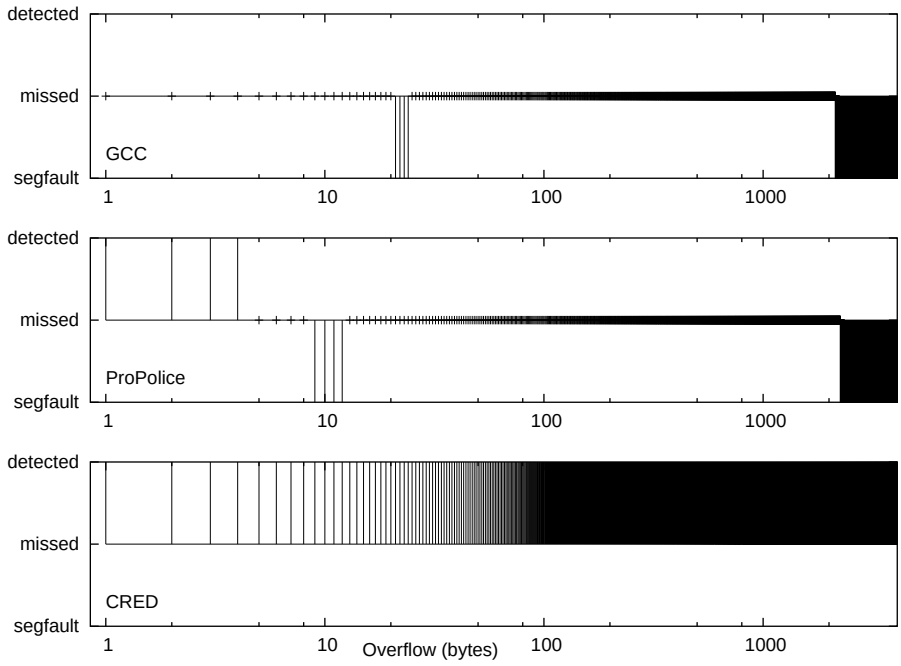

Figure 4-3: Results for *discrete* buffer overflow example. *The example consists of a single out-of-bounds write (see Figure 4-1 for source code). Results are shown for ProPolice, GCC and CRED.*

```
1  #define BUFF_SIZE 200
2
3  // generates null-terminated heap buffer
4  char *generate_buffer(int size);
5
6  typedef struct {
7      char buf[BUFF_SIZE];
8      int val;
9  } my_struct;
10
11 int main(int argc, char **argv) {
12     my_struct m;
13     char *bufB;
14     int delta = atoi(argv[1]);
15     bufB = generate_buffer(BUFF_SIZE + delta);
16     assert(bufB != NULL);
17
18     strcpy(m.buf, bufB);
19
20     return 0;
21 }
```

Figure 4-4: Example of *continuous* overflow test case

overflow, ranging from 1 to 4096 bytes. As before, for each amount of overflow in bytes, *detection* is represented by an upward bar, a *miss* is shown by a hash mark on the middle line and a *segfault* is indicated by a downward bar.

The program compiled with gcc gives no indication that an overflow is occurring until part of the return instruction pointer (IP) is overwritten. Since the return IP is also overwritten for all larger overflows, they all result in a segfault. Similar to the discrete case, very large overflows cause a segfault by referencing memory beyond the stack page boundary.

This test case is essentially the classic "stack smashing" attack, so ProPolice performs well. It detects most of the overflows, since they modify the "canary" value. The first overflows of size 1–4 bytes correspond to overwriting the integer member of the struct (`m.val`), and they remain undetected, since the "canary" is placed above the struct in memory. Large overflows reveal another limitation of ProPolice – since the "canary" value is checked only when returning from a function, a large overflow may cause a segfault before it is detected.

CRED detects all overflows that reference memory outside of the struct. As discussed in Section 4.1.3, CRED treats all members of the struct as belonging to a single object. This limitation allows overwriting of `m.val` to remain undetected. There seems to be no

simple fix for this issue that would not create false alarms, as arbitrary casts between object pointers are allowed by the C standard [46].

4.2.3 Variable-overflow Testsuite Results

This section presents a summary of the results obtained with the variable-overflow testsuite. The graph in Figure 4-7 shows the fraction of test cases in the variable-overflow testsuite with a non-*miss* (*detection* or *segfault*) outcome for each amount of overflow. Higher fractions represents better performance. All test cases, with the exception of the 4 underflow test cases, are included on this graph even though the proportional composition of the testsuite is not representative of existing exploits. Nonetheless, the graph gives a good indication of tool performance. Fine-grained bounds checking tools are highlighted by the "fine-grained" box at the top of the graph.

The top performing tools are Insure++, CCured and CRED, which can detect small and large overflows in different memory locations. TinyCC also performs well on both heap and stack-based overflows, while ProPolice only detects continuous overflows and small discrete overflows on the stack. Since the proportion of stack-based overflows is larger than that of heap-based overflows in our testsuite, ProPolice is shown to have a relatively high fraction of detections. Chaperon and Valgrind follow the same shape as gcc, since these tools only provide fine-grained detection of overflows on the heap. This ability accounts for their separation from gcc on the graph.

As the graph demonstrates, only tools with fine-grained bounds checking, such as Insure++, CCured and CRED are able to detect small overflows, including off-by-one overflows, which can still be exploitable. For tools with coarse stack monitoring, a large increase in detections/segfaults occurs at the overflow of 21 bytes, which corresponds to overwriting the return instruction pointer. The drop after the next 4 bytes corresponds to the discrete overflow test cases, as they no longer cause a segfault behavior. ProPolice exhibits the same behavior for overflows of 9–12 bytes due to a slightly different stack layout. Tools with fine-grained bounds checking also perform better in detecting discrete overflows and thus do not exhibit these fluctuations. For very large overflows, all tools either detect the overflow or segfault, which results in fraction of non-*miss* results close to 1, as shown on the far right side of the graph.

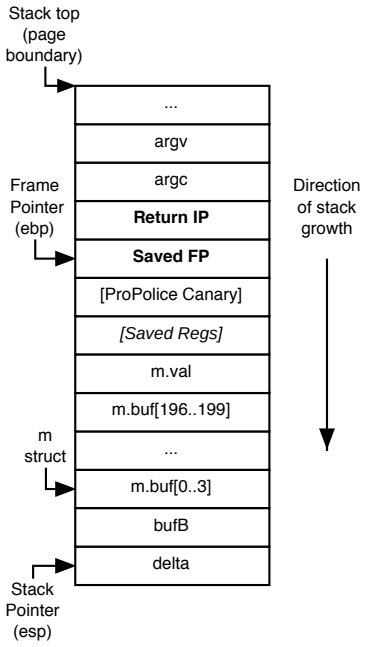

Figure 4-5: Stack layout for *continuous* buffer overflow example. *This example consists of a single out-of-bounds write (see Figure 4-4 for source code). ProPolice "canary" is present only in the ProPolice test case, and saved registers appear only in GCC and CRED test cases.*

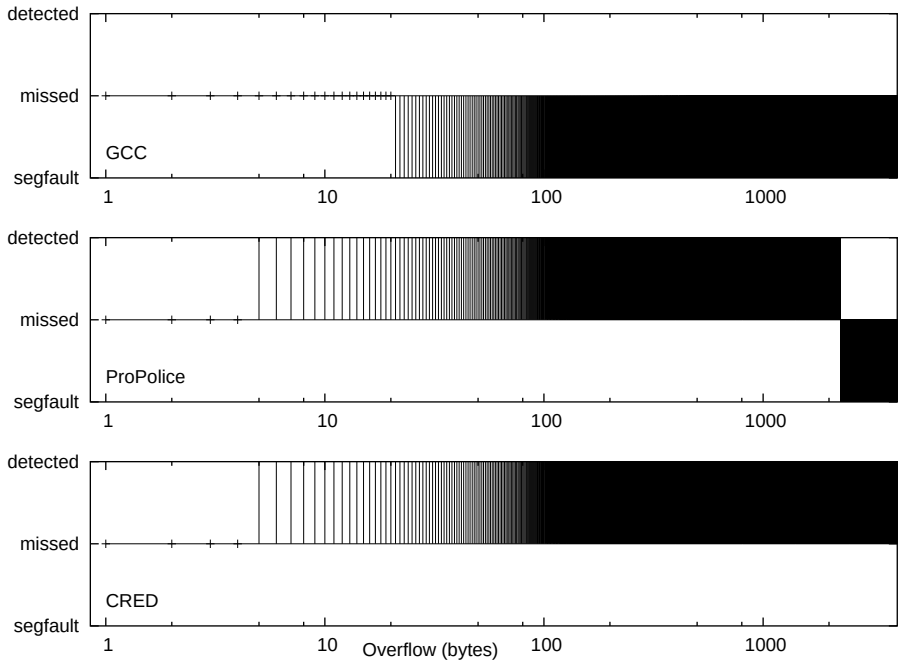

Figure 4-6: Results for *continuous* buffer overflow example. *This example consists of a single out-of-bounds write (see Figure 4-4 for source code). Results are shown for ProPolice, GCC and CRED.*

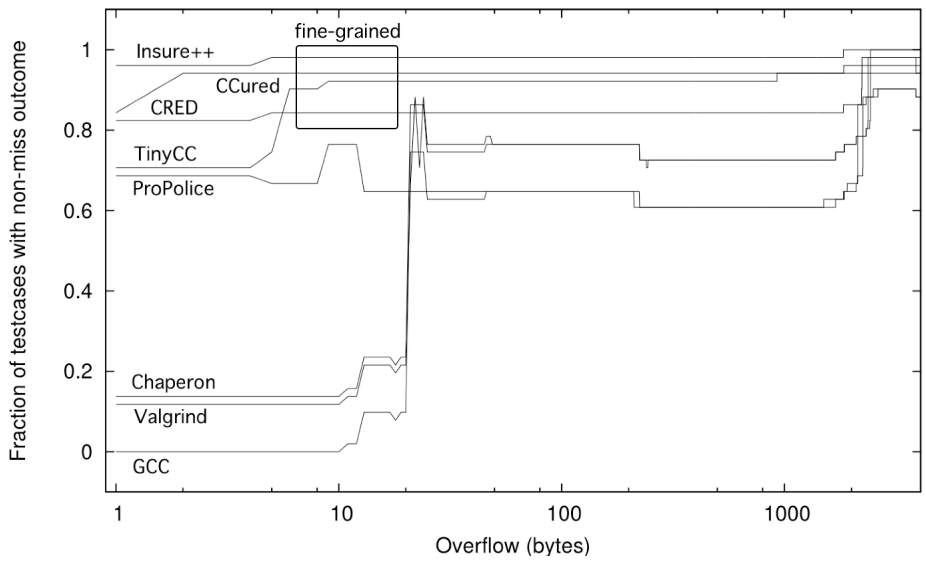

Figure 4-7: Summary of results for variable-overflow testsuite. *The graph shows combined fraction of detections and segfaults vs the amount of overflow in bytes. A box highlights tools with fine-grained bounds checking capabilities.*

4.3 Artificial Exploit Evaluation

The *artificial exploit* testsuite is the second in the series of three evaluations presented in this chapter. The test cases in the artificial exploit testsuite are adapted from Wilander's evaluation of dynamic buffer overflow detection tools [59]. Unlike the variable-overflow testsuite, these test cases actually exploit the buffer overflow to produce a shell. The goal of this evaluation is to test each tool's ability to detect or prevent an exploit based on a buffer overflow.

The *artificial exploit* testsuite consists of 18 test cases, each of which uses a 16-element `long` buffer located either on the stack or in the bss section. The attacks target different elements of the program that allow for change of control flow. A classic "stack smashing" attack overwrites the return instruction pointer (IP) to point to assembly code that produces a shell once the current function returns. Another attack overwrites the saved frame pointer (FP) to point to a carefully crafted stack frame that includes a return IP that changes program flow. Other attacks target function pointers, and `longjmp` buffers to spawn a shell. Each target is overwritten either directly (by overflowing the buffer all the way to the target) or indirectly (by overflowing the buffer to change a pointer, which is then used to change the target).

4.3.1 Test Procedure

The artificial exploit test cases were compiled and executed with each tool. The result is one of four possible outcomes – *prevented*, indicating that the attack has been halted and an error message is printed, *detected*, signifying that an error message is printed, but the attack is not stopped, *segfault*, indicating either a detection (in TinyCC) or an illegal memory access, and *miss*, indicating that the attack succeeded and resulted in a spawned shell.

4.3.2 Artificial Exploit Results

In Wilander's artificial exploits, the tools with fine-grained bounds checking prevented all the exploits. ProPolice was able to prevent some attacks due to rearranging of local variables such that function pointers were protected. Chaperon and Valgrind detected 28% and 11% of attacks, respectively, allowing the rest to proceed. Overall, only fine-grained bounds checking tools were able to reliably prevent the attacks.

These results are consistent with the results from the variable-overflow testsuite. Most of these attacks involve a continuous write that overflows a buffer, which was tested by the variable-overflow testsuite. Using a `long` buffer makes the attacks harder to detect for some tools, such as ProPolice, since it only protects functions with `char` buffers.

4.4 Real Exploit Evaluation

As noted at the beginning of this chapter, the work described in this section has been performed by Tim Leek. The text is replicated for continuity from "Dynamic Buffer Overflow Detection" by Michael Zhivich, Tim Leek and Richard Lippmann [61]. This paper has been submitted to FSE 2005.

Previously, we evaluated the ability of a variety of tools employing *static analysis* to detect buffer overflows [63]. These tools ranged from simple lexical analyzers to abstract interpreters [52, 18, 22, 56, 60]. We chose to test these tools against fourteen historic vulnerabilities in the popular Internet servers `bind`, `sendmail`, and `wu-ftpd`. Many of the detectors were unable to process the entire source for these programs. We thus created *models* of a few hundred lines that reproduced most of the complexity present in the original. Further, for each model, we created a patched copy in which we verified that the overflow did not exist for a test input that triggered the error in the unpatched version. In that

	Chaperon	Valgrind	CCured	CRED	gcc	Insure	ProPolice	TinyCC
b1		d	d		d			d
b2		d	d		d			d
b3			d	d	d	d	d	d
b4	df	df	d	d	d	df	d	df
f1			d	d		d		d
f2			d	df		df		d
f3			d	d		d		d
s1			d	d		d		d
s2			d	d		df		d
s3			d	d				d
s4			d	d				d
s5			d	d	d	df	d	d
s6			df	d		d		
s7	d	d		d	d	d		d
$P(det)$	0.14	0.29	0.93	0.86	0.43	0.71	0.21	0.93
$P(fa)$	0.07	0.07	0.07	0.07	0	0.29	0	0.07

Table 4.2: Dynamic buffer overflow detection in 14 models of real vulnerabilities in open source server code. *There are four* `bind` *models (b1–b4), three* `wu-ftpd` *models (f1–f3), and seven* `sendmail` *models (s1–s7). A 'd' indicates a tool detected a historic overflow, while an 'f' means the tool generated a false alarm on the patched version.* $P(det)$ *and* $P(fa)$ *are the fraction of model programs for which a tool signals a detection or false alarm, respectively.*

evaluation, we found that current static analysis tools either missed too many of these vulnerable buffer overflows or signaled too many false alarms to be useful. Here, we report results for seven dynamic overflow detectors on that same set of fourteen models of historic vulnerabilities. This provides a prediction of their performance on real overflows that occur in open-source servers.

4.4.1 Test Procedure

During testing, each unpatched model program was compiled with the tool (if necessary) and executed on an input that is known to trigger the overflow. A *detection* signifies that the tool reported an overflow, while a *miss* indicates that the program executed as if no overflow occurred. A patched version of the model program was then executed on the same input. A *false alarm* was recorded if the instrumented program still reported a buffer overflow.

4.4.2 Real Exploit Results

Table 4.2 gives the results of this evaluation, which agree well with those on the variable-overflow testsuite. Three of the dynamic overflow detectors that provide fine-grained bounds checking, CCured, CRED, and TinyCC, work extremely well, detecting about 90% of the overflows whilst raising only one false alarm each. The commercial program Insure, which also checks bounds violations rigorously, fares somewhat worse with both fewer detections and more false alarms. Notice that misses and false alarms for these tools are errors in the implementation, and are in no way a fundamental limitation of dynamic approaches. For example, in the case of CRED the misses are due to an incorrect `memcpy` wrapper; there are no misses once this wrapper is corrected. The CRED false alarm is the result of overly aggressive string length checks included in the wrappers for string manipulation functions such as `strchr`. None of the tools are given credit for a segmentation fault as a signal of buffer overflow (except TinyCC and gcc as this is the only signal provided). This is why, for instance, ProPolice appears to perform *worse* than gcc. As a final comment, it is worth considering the performance of gcc alone. If provided with the right input, the program itself detects almost half of these real overflows, indicating that input generation may be a fruitful area of future research.

4.5 Performance Overhead

The goals of the performance overhead evaluation are two-fold. One is to quantify the slowdown caused by using dynamic buffer overflow detection tools instead of gcc when executing some commonly used programs. The other is to test each tool's ability to compile and monitor a complex program. In addition, this evaluation shows whether the tool can be used as a drop-in replacement for gcc, without requiring changes to the source code. Minimal modifications to the makefile are allowed, however, to accommodate the necessary options for the compilation process.

Our evaluation tests overhead on two common utility programs (`gzip` and `tar`), an encryption library (`OpenSSL`) and a webserver (`Apache`). For `OpenSSL` and `tar`, the testsuites included in the distribution were used. The test for `gzip` consisted of compressing a tar archive of the source code package for glibc (around 17MB in size). The test for `Apache` consisted of downloading a 6MB file 1,000 times on a loopback connection. The overhead

was determined by timing the execution using `time` and comparing it to executing the test when the program is compiled with gcc. The results are summarized in Table 4.3. Programs compiled with gcc executed the tests in 7.2s (`gzip`), 5.0s (`tar`), 16.9s (`OpenSSL`) and 38.8s (`Apache`).

Tool	**gzip**	**tar**	**OpenSSL**	**Apache**
Chaperon	75.6		61.8	
Valgrind	18.6	73.1	44.8	
CCured				
CRED	16.6	1.4	29.3	1.1
Insure++	250.4	4.7	116.6	
ProPolice	1.2	1.0	1.1	1.0
TinyCC				

Table 4.3: Instrumentation overhead for commonly used programs. *The overhead is presented as a multiple of gcc execution time. The blank entries indicate that the program could not be compiled or executed with the corresponding tool.*

Compiling and running `Apache` presented the most problems. Chaperon requires a separate license for multi-threaded programs, so we were unable to evaluate its overhead. Valgrind claims to support multi-threaded programs but failed to run due to a missing library. Insure++ failed on the configuration step of the makefile and thus was unable to compile Apache. CCured likewise failed at configuration, while TinyCC failed in parsing one of the source files during the compilation step.

The performance overhead results demonstrate some important limitations of dynamic buffer overflow detection tools. Insure++ is among the best performers on the variable-overflow testsuite; however, it incurs very high overhead. CCured and TinyCC, which performed well on both the variable-overflow testsuite and the model programs, cannot compile these programs without modifications to source code. CCured requires the programmer to annotate sections of the source code to resolve constraints involving what the tools considers "bad casts," while TinyCC includes a C parser that is likely incomplete or incorrect.

While CRED incurs large overhead on programs that involve many buffer manipulations, it has the smallest overhead for a fine-grained bounds checking tool. CRED can be used as a drop-in replacement for gcc, as it requires no changes to the source code in order to compile these programs. Only minimal changes to the makefile were required in order to enable bounds-checking and turn off optimizations.

4.6 Discussion

The three top-performing tools in our evaluation are Insure++, CCured and CRED. Insure++ performs well on test cases, but not on model programs. It adds a large performance overhead and the closed-source nature of the tool inhibits extensions. CCured shows a high detection rate and is open-source; however, it requires rewriting 1–2% of code to compile complicated programs [10]. CRED also offers a high detection rate, and it is open-source, easily extensible and has fairly low performance overhead (10% slowdown for simple Apache loopback test). Its main disadvantage is not detecting overflows in library functions compiled without bounds-checking.

As this study demonstrates, several features are crucial to the success of a dynamic buffer overflow detection tool. Memory monitoring must be done on a fine-grained basis, as this is the only way to ensure that discrete writes and off-by-one overflows are caught. Buffer overflows in library functions, especially file I/O, often go undetected. Some tools solve this problem by creating wrappers for library functions, which is a difficult and tedious task. Recompiling libraries with the bounds-checking tool may be a better alternative, even if it should entail a significant slowdown. Error reporting is likewise essential in determining the cause of the problem because segfaults alone provide little information. Since instrumentation and messages can get corrupted by large overflows, the error should be reported immediately after the overflow occurs.

Of all the tools surveyed, CRED shows the most promise as a part of a comprehensive dynamic testing solution. It offers fine-grained bounds checking, provides comprehensive error reports, compiles large programs and incurs reasonable performance overhead. It is also open-source and thus easily extensible. CRED is likewise useful for regression testing to find latent buffer overflows and for determining the cause of segfault behavior. For these reasons, CRED is used to provide code instrumentation to detect buffer overflows in the dynamic testing and adaptive testing systems presented in the following chapters.

Chapter 5

Dynamic Testing with Code Instrumentation

As discussed in Chapter 3, several systems have been developed to test software automatically. The state of the art system in mini-simulation and automatic software testing has been developed at the University of Oulu, Finland as part of the PROTOS project [28]. This system utilizes an attribute grammar based on BNF to describe possible inputs, including deviations from the protocol, such as invalid checksums or misplaced elements. Random inputs are then created by a generator based on this grammar. The program is executed on resulting inputs, and overflow is detected by observing whether the program is hung or has segfaulted.

A major shortcoming of the PROTOS approach is the use of segfaults for buffer overflow detection. Small write overflows and most read overflows do not cause a segfault, and, as discussed in Chapter 2, even small overflows can be exploited; thus, the PROTOS system may miss important overflows that can enable attacks. Another disadvantage of using segfaults as an indicator of an overflow is lack of information about the error that has occurred. Segfaults merely indicate that an attempt has been made to access memory that does not belong to the process. They provide no information about where in the program the error has occurred, which is necessary to correct the fault. While such information can also be obtained from a core dump, extracting it is a much more difficult and tedious process.

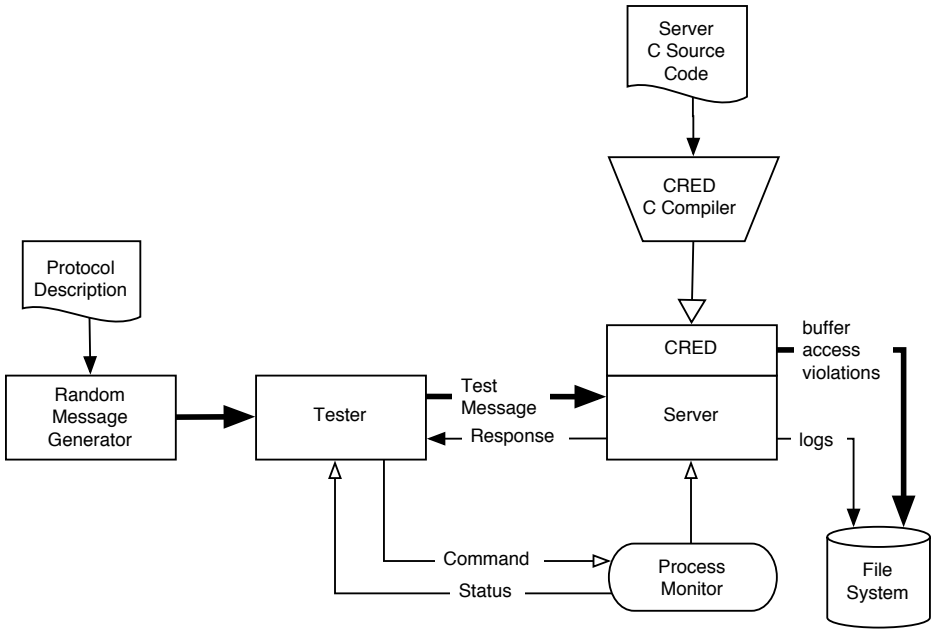

Figure 5-1: System architecture for dynamic testing with code instrumentation. *Thick black arrows indicate the movement of data through the system. Dashed lines indicate control flow.*

To improve on the results of the PROTOS system, the new approach developed in this thesis uses compiler-based code instrumentation that provides fine-grained bounds checking to determine whether an overflow has occurred. In addition, two alternative methods for generating random messages are explored.

5.1 Dynamic Testing Architecture

Figure 5-1 presents the architecture of a testing system that uses random messages and code instrumentation to detect buffer overflows. This architecture has been developed for testing server software that accepts input messages; however, it could be easily adapted to test other kinds of software. The source code for the server is compiled using the CRED compiler and test messages are provided by the *random message generator*. The *tester* module is responsible for coordinating message generation and test case execution. A *process monitor* is used by the tester to restart the server if it should be terminated by instrumentation due to a detected overflow or quit unexpectedly for some other reason. Each of the modules in this diagram is described further in the following sections.

5.1.1 Code Instrumentation

As demonstrated in the evaluation presented in Chapter 4, CRED is capable of providing fine-grained buffer overflow detection in all memory areas. It is also capable of compiling large programs, which is essential in order to provide code instrumentation for server software. Using CRED to provide buffer overflow detection promises to yield more complete results than those obtained in the PROTOS project, as buffer overflow detection of the resulting system is much more fine-grained. CRED likewise addresses another problem posed by using segfaults for overflow detection — it provides detailed information about where the overflow has occurred, including the file and line of the instruction causing an overflow as well as the location of the object's creation. Such information makes it easier for the developer to locate and fix mistakes.

One of the disadvantages of using CRED as a compiler is that tests will experience a slowdown due to the performance overhead caused by checking each buffer access. However, as discussed in Chapter 4, such overhead, while at times significant, is an acceptable price for a much more comprehensive monitoring of memory accesses. Another disadvantage of this method is that source code is required for testing; however, this is necessary for fine-grained buffer overflow detection, and this system is designed for use by developers of the program under test.

5.1.2 Tester and Random Message Generator

The tester module is responsible for coordinating production and execution of test cases on the server. It emulates a client that understands the server's protocol to set up a session (if it is required), sends test messages to the server, and handles server shutdowns and session terminations. The tester accepts configuration options that specify timeouts, number of tests to be executed, and a seed for the random message generator. The seed is needed to make test runs repeatable and deterministic. The tester also can keep track of statistics for each test run, including the time for message processing, number of messages sent per session, number of sessions, etc.

The random message generator accepts a protocol description and a seed and generates messages that conform to the given protocol description. Such description should include deviations from the standard protocol accepted by the server in order to generate messages

that will stress robustness and error-handling capabilities of the server. The message generation can be done online during the testing or offline, since the parameters of a test do not change during a test run. Random message generators developed for this system are described in Section 5.2.

5.1.3 Process Monitor

The PROTOS system required a process monitor because the program under test was likely to become unstable if an overflow were discovered. In a favorable case, the program would segfault; however, it could also hang waiting for something to happen or go into an infinite loop. In these cases, a process monitor was used to shutdown and restart the server. Since CRED is used for buffer overflow detection in the system presented here, program corruption is less likely, and an overflow will result in a graceful termination of the program. However, some messages may cause the server to shutdown abnormally, which necessitates having facilities that allow the server to be automatically restarted. Another reason for creating a process monitor is test performance, as the timeouts used by the testing script to wait for an answer from the server can be greatly reduced, since another mechanism is available to determine whether the server is running.

The process monitor is implemented as a Perl script that sets up the proper environment variables and spawns the server process (or multiple processes if the server requires more than one process). The process monitor accepts commands from the tester module over a socket, allowing the tester to query server status and issue commands to start and shutdown the server. Communicating over a socket enables the server and process monitor to run on a separate computer from the tester and message generator, which can be useful if the server or message generator require extensive resources. The server process is controlled by the process monitor through signals — a `SIGKILL` is sent to the server to terminate it, and the process monitor receives `SIGCHLD` whenever the server process terminates. In order to determine server status, the process monitor calls `kill 0` with the server's PID. While this does not send any signals, it does verify whether the PID is valid, thus indicating whether the server is still running.

5.2 Random Message Generation

As discussed in Chapter 3, a system that uses dynamic testing requires a *revealing input* that will trigger the overflow in order for the overflow to be detected. Generating random messages is one of the techniques available for exploring the input space of a program and exercising its ability to process different inputs. This method is particularly attractive for large input spaces when enumerating all possible inputs is not feasible. Furthermore, protocol descriptions for random message generators enable the developer to choose which portions of the input space will be explored randomly and which portions will be fixed. This makes it possible to focus on particular parts of the program under test; however, it requires a good understanding of code paths as well as additional effort on the part of the developer to create such protocol descriptions. Two different methods of describing and generating random messages are discussed in this section: a Hidden Markov Model (HMM) and Probabilistic Context Free Grammar (PCFG).

5.2.1 HMM Random Message Generator

The Hidden Markov Model (HMM) random message generator is based on the concept that input messages are a sequence of string tokens. Generating these messages can be described by an HMM [47], in which each state produces a token that will appear in the message, and state transitions determine the order in which such tokens appear. Each state contains a multinomial model that defines a discrete probability distribution over tokens, one of which is picked at runtime. Tokens are specified by a character set that describes their composition, a length, and a generator function.

The HMM random message generator used in this project was implemented in Perl. A description of the HMM is written in XML for easy readability and parsing. An example HMM description and an explanation of the syntax can be found in Appendix A. The HMM description defines *states* that contain a multinomial probability distribution over *tokens*, *state transitions* that specify source and destination states as well as the probability of the transition and a *start state distribution* that defines a probability of starting in a particular state. A *token* description specifies its length, a character set that determines its composition and a name of the generator function.

At runtime, the generator picks a *start state* according to the probability distribution specified by *start state distribution*. A token is picked according to the distribution in that state and generated using the specified generation function. The *charset* specified for this token is used for the token's composition, and the length determines how many characters the token will contain. The length can be a constant, or can be specified as a *stop probability*. Using a stop probability leads to a geometric distribution of lengths, since the average length of the string generated with stop probability p is $\frac{1}{p}$. Once the token is constructed, the message generator determines whether the current state is an *end state* (a state with no outgoing transitions). If it is, the entire message has been constructed. Otherwise, a transition to the next state is made according to the specified distribution, and the process repeats.

One of the main advantages of this approach is that paths through the message generator are easy to visualize, as an HMM description lends itself easily to a graphical representation. Another advantage is that token generators are defined as Perl functions, thus allowing for arbitrary division of the message into tokens, since complexity can be shifted between the XML description and Perl generator functions. This makes it possible to isolate different sections of the input message and create arbitrarily complex test cases using Perl generators.

Due to implementation in Perl, the HMM random message generator is somewhat slow. Creating a detailed HMM description is also rather tedious, since XML is a verbose language. Because the HMM definition is not hierarchical, elements may need to be repeated, making the description redundant in some cases; this problem is somewhat mitigated by utilizing XML entities that enable insertion of replacement XML code in documents. However, these limitations apply more to the implementation of the HMM random message generator than to the approach in general.

5.2.2 PCFG Random Message Generator

An alternative method of describing a protocol for random message generation is a Probabilistic Context Free Grammar (PCFG). Such a grammar can be based on a BNF description of the protocol used by the server for communication. BNF provides a concise hierarchical description of the protocol, and although it is generally used to validate messages against the protocol, it can also be used to generate messages. The probabilistic nature of this grammar manifests itself in assigning probabilities to the choices and options presented by the BNF protocol description.

The PCFG description consists of productions that define a hierarchical structure of elements that are concatenated together. An example PCFG and an explanation of the syntax can be found in Appendix B. PCFG descriptions utilize the following constructs to describe the protocol:

start A starting production that defines the root of the BNF tree.

terminal A node consisting of literal characters that will appear in the final output.

non-terminal A node consisting of a concatenation of *terminals* or other *non-terminals*

choice A production may indicate that one of the *terminals* or *non-terminals* from the given list should be chosen by separating the list with a vertical bar (`|`). The discrete probability for choosing each option is provided as part of the PCFG description.

option Enclosing a *terminal* or *non-terminal* in square brackets (`[]`) makes the item optional. A probability of exercising this option is specified as part of PCFG description.

one-or-more Enclosing a *terminal* or *non-terminal* in curly braces ({}) specifies that it should be repeated. The number of occurrences is distributed according to a normal distribution, the mean and variance for which are specified as part of the PCFG.

The PCFG generator works by starting at the *start* production, and instantiating its elements, subject to repetition and option rules and corresponding probabilities. For each element, the generator finds the production where the element appears on the left-hand side, replaces it with the right-hand side of the production, and instantiates the resulting elements, again following repetition rules with appropriate probabilities. The process continues recursively until all non-terminals have been expanded into literal characters, which are concatenated together to create the resulting message.

The PCFG generator used in this project is based on a generator developed by Berke Durak to create random messages based on a BNF [15]. This generator was implemented in OCaml, and required extensions to support PCFG descriptions. Tim Leek kindly volunteered his time and expertise to make it work. The resulting generator accepts a description of the protocol in an extended BNF format, with probabilities for choices or options provided on the line after the production that uses them. Productions define different components of the input message and can be arbitrarily detailed. Unlike the HMM description, all the complexity of the message is described in the PCFG format, as the random message generator does not provide facilities for specifying token generators in OCaml; however, the

hierarchical structure of PCFG allows for a more concise specification than the XML format used for HMM descriptions.

The advantage of using the PCFG generator is that a BNF description of the protocol for the server under test often already exists, and a PCFG can be obtained by assigning desired probabilities and other appropriate parameters to different choices and repetitions. A disadvantage of both the PCFG and the HMM approaches, is that semantic dependencies between message parts that are present in human-readable protocol descriptions are lost during the translation. For example, a BNF can specify that `len` is an integer and `str` is a string, but it has no simple facility to specify that `len` is actually the length of `str`. Thus, messages generated by both the PCFG and the HMM random message generators may not be semantically correct, though they will be syntactically correct.

5.3 Evaluating Dynamic Testing

As mentioned before, servers face a larger share of scrutiny and attacks, since they usually represent targets that contain information of interest to crackers or present a financial asset to the victim. The technologies discussed in this thesis are thus geared towards helping developers create more secure server software. The dynamic testing approach presented in this chapter was evaluated using an implementation of Jabber Instant Messaging server. The server, testing platform and evaluation results are described in detail in the remainder of the section.

5.3.1 Jabber Instant Messaging Server

The server tested in this evaluation is a C implementation of Jabber Instant Messaging Server, `jabberd-2s03` [39]. Jabber is an open instant messaging protocol based on XML streams, that "enables any two entities on the Internet to exchange messages, presence and other structured information in close to real time" [20]. Jabber offers several key advantages over other instant messaging systems — the protocols are free, open, public and have been approved by the Internet Engineering Task Force [23], the structure of the service is decentralized, and XML allows for extensible and flexible messaging.

A typical Jabber server deployment is shown in Figure 5-2. Rectangular boxes represent servers, while rounded tags represent users. Lines with black filled arrows show the

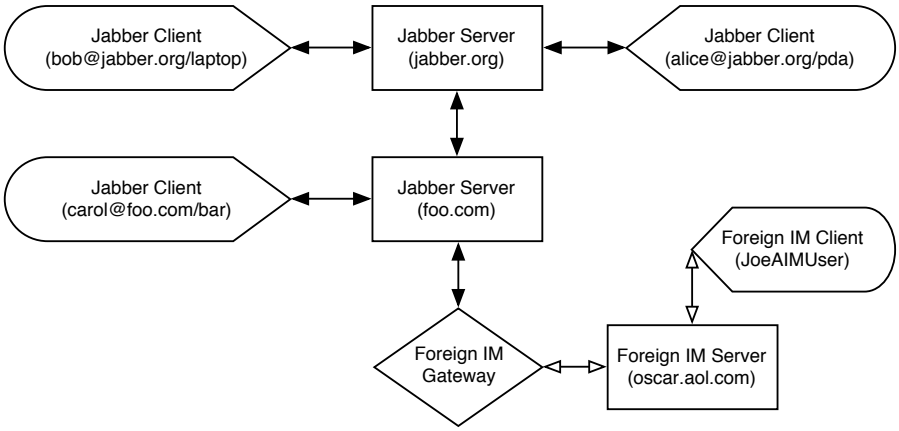

Figure 5-2: Typical Jabber server deployment. *Rectangular boxes represent servers, while rounded tags represent users. The diamond box represents a gateway that translates messages between Jabber and a proprietary messaging format. Arrows show the flow of messages between servers and clients.*

movement of Jabber messages, while lines with unfilled arrows show the flow of messages in some foreign instant messaging protocol. As demonstrated in the diagram, each user is affiliated with some Jabber server, to which he connects in order to send messages. These messages can be sent to Jabber users connected to another server via server-to-server communication. In order to communicate with users of other instant messaging systems, such as AIM [41] or Yahoo! Messenger [24], gateways that translate between Jabber and foreign instant messaging protocols have been created. While such gateways are not part of the Jabber server, they are available as third-party software.

Jabber uses a scheme that is very similar to e-mail to locate and identify its users. Each client has a Jabber ID (JID) that has the form of `user@host/resource`. The `host` identifies the Jabber server onto which the `user` logs on (that server stores the user's authentication information and contact list). The `resource` makes it possible to differentiate multiple connections from the same user — for example, one may be logged in from a desktop in the office and from a laptop in the conference room simultaneously. A priority system exists to notify the server where the messages are to be forwarded if the user is logged in from multiple locations. Having multiple decentralized servers allows Jabber to be scalable, since no single server needs to bear the brunt of storing authentication information and processing messages for all users. In addition, this design makes it possible for a company to run an internal Jabber server and keep that traffic confidential while still allowing employees to

carry on conversations with outside Jabber users.

Both humans and computer programs can use Jabber services to exchange messages, and gateways have been built to allow Jabber IM to interact with other IM services that may have proprietary protocols. Most of the complexity for processing Jabber messages is relegated to the server, which allows for easy creation of light-weight clients. Such clients exist for almost any platform in a variety of languages. Since the server is the complex component implemented in C, it is well suited for buffer overflow testing.

Figure 5-3 presents the architecture of the `jabberd` server. MySQL database and the Foreign IM Gateway are shown in the diagram for completeness, but are not included in `jabberd` server. While the intricate details are not important for the purposes of this evaluation, it is worthwhile noting that the server is designed in a scalable manner: it consists of five different processes that exchange information over TCP connections; thus, each process can run on a separate computer to distribute the load. Each of the processes is responsible for a different part of message processing:

router The *router* process is the backbone of the `jabberd` server. It accepts connections from other server components and passes XML packets between appropriate components.

s2s The *server to server* component handles communications with external servers. It passes packets between other components and external servers, and also performs dialback to authenticate remote Jabber servers.

resolver The *resolver* process acts in support of `s2s` component. It resolves hostnames as part of dialback authentication.

sm The *session manager* component implements the bulk of instant messaging features, including message passing, presence information, rosters and subscriptions. This component connects to the authentication/data package to provide persistent data storage.

c2s The *client to server* module interacts with the Jabber client and passes packets to `sm` through the `router`. It also handles client authentication and registration.

5.3.2 Test Environment

The grammar-based dynamic testing environment for `jabberd` is implemented according to the architecture shown in Figure 5-1. The messages are generated from a protocol description by a random message generator, such as the HMM or PCFG generators discussed in this chapter. The tester coordinates sending these messages to `jabberd` and restarting the

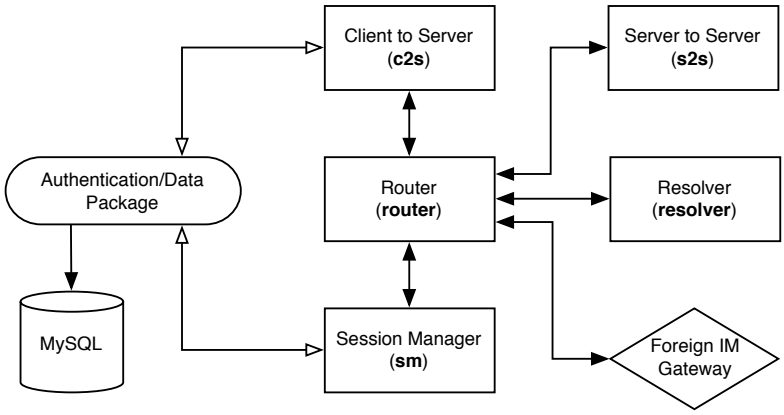

Figure 5-3: Architecture of `jabberd`. *MySQL and Foreign IM Gateway are not part of the `jabberd` server*

server if necessary. Code instrumentation is provided by the CRED C compiler that inserts code to check for bounds violations and halt the program should a buffer access violation occur. In addition, CRED provides bounds error reports that inform the developer about the location of the error. The process monitor is used to facilitate automatic server restarts and status reporting.

As mentioned before, `jabberd` consists of five different processes; however, it was tested as a single server through the interface presented to a Jabber client. The process monitor was responsible for setting up the appropriate environment for each of the `jabberd`'s processes and starting them up in the proper order so that all connections could be made. Each process had a slightly different environment that allowed CRED to log buffer access violations into a separate file for each process. The messages, server logs and CRED buffer access violations were stored on the file system during these tests. All processes, including the tester and `jabberd` processes, ran on the same computer, though, as mentioned before, the components communicate via TCP sockets and thus could be distributed among different hosts. The computer used in testing is a dual 2.66GHz Xeon CPU machine running Red Hat Linux release 9 (Shrike).

5.3.3 Test Procedure

The Jabber server was compiled with CRED C compiler, and the executable was tested using the framework shown in Figure 5-1. Protocol descriptions for both PCFG and HMM random

```
<message
  to='alice@swordfish.ll.mit.edu/resource'
  from='bob@swordfish.ll.mit.edu/test'
  id='12345'>
    <thread>
      1234567890
    </thread>
    <subject>
      This is a sample subject.
    </subject>
    <body>
      This is a sample body.
    </body>
</message>
```

Figure 5-4: A typical Jabber IM message. *Sender, recipient and message id are specified as attributes of the* `<message/>` *XML element. The* `<thread/>` *element is used by Jabber clients to group messages, while contents of* `<subject/>` *and* `<body/>` *XML elements are presented to the user.*

message generators were constructed to generate messages that were slightly malformed in various ways. A typical Jabber IM message is shown in Figure 5-4. Some examples of messages created by the PCFG random message generator can be found in Appendix D.

From reading the protocol description of Jabber messages and XML documents, the characters listed in Table 5.1 were found to be important in separating different elements of the message. These delimiters will be referred to as *control characters* throughout the remaining discussion. The protocol descriptions for HMM and PCFG generators were designed with this knowledge in mind to perform the following tests:

length test The messages used in this test adhered to the correct protocol grammar; however, the lengths of all value strings were distributed normally with mean of 1024 and variance of 1024, treating negative values as 0. The mean and variance values were chosen to produce messages that can overflow buffers of 1024 and 2048 bytes.

missing/wrong control characters test The messages used in this test were similar to those in the *length test*; however, control characters were sometimes missing or incorrect.

missing/duplicated attributes test The messages used in this test were similar to those in the *length test*; however, some XML attributes were either missing or duplicated.

The goal of these tests was to explore various error conditions that may occur in the Jabber server. The *length test* checks for proper bounds calculations for buffers used to parse

Control Character	Function
`<`	Opens start and end XML tags
`>`	Closes start and end XML tags
`space`	Separates tag attribute-value pairs
`=`	Separates tag attributes and values
`'`	Encloses values of tag attributes
`@`	Separates `user` and `host` portions of JID
`/`	Separates `host` and `resource` portions of JID (also used in opening end XML tag)

Table 5.1: Control characters in Jabber messages. *The angle brackets delineate XML message elements, spaces separate attribute key-value pairs of XML elements. The '@' and '/' characters are used to separate different portions of JID.*

the input. The *missing/wrong control characters test* checks for possible type confusion due to a wrong interpretation of some element. The *missing/duplicated attributes test* checks for server behavior when encountering an unexpected token or not receiving a token that was expected.

A single PCFG and an equivalent HMM description were created to generate the messages for all three tests. The HMM and PCFG protocol descriptions used in these tests are shown in Appendix A and Appendix B, respectively. The PCFG description is 60 lines long and consists of 39 productions, whereas the HMM description is 492 lines long and contains 59 states and 105 state transitions.

Using these protocol descriptions 10,000 messages were generated and then sent to the `jabberd` server. Whenever a bounds error was reported or the server quit for some other reason, the process monitor was used to restart the server, and the tester continued sending messages. The log files generated by CRED were then examined to determine whether buffer access violations occurred during the test. The corresponding messages that caused these violations were then recovered from tester logs. For each overflow found, the message causing the overflow was then sent to a `jabberd` server compiled with gcc to determine whether it causes the program to segfault or hang without CRED instrumentation.

5.3.4 Test Results

The overflows revealed by grammar-based testing are shown in Table 5.2. The table details the location of the access violation, the location of buffer creation and the memory area where it is stored. All this information is obtained directly from CRED instrumentation,

Process	Access Violation	Object	Created at	Storage	Type	Message Count	Detected by Segfault
`s2s`	main.c: 129	`secret`	main.c: 65	stack	write	1	No
`sm`	jid.c: 103	`str`	jid.c: 81	stack	write	18	Yes
`sm`	jid.c: 115	`str`	jid.c: 81	stack	write	26	No
`sm`	jid.c: 127	`str`	jid.c: 81	stack	write	14	Yes

Table 5.2: Overflows detected in grammar-based testing. *Message count shows the number of times the overflow was found during 10,000 message test run. The overflow is "detected by segfault" if a version of* `jabberd` *compiled with gcc segfaults on the message that causes this overflow.*

with no further developer effort required. The actual error messages generated by CRED are presented in Appendix C. The messages causing these overflows can be found in Appendix D. The table also lists the results of sending these messages to a `jabberd` server compiled with gcc. The *message count* field in the table shows the number of times a particular overflow was found during the 10,000 message test run that lasted approximately 35 minutes.

The overflow in `s2s` occurs during parsing of the configuration file at startup because of a programmer error — a `NULL` string terminator is written to `secret[40]`, which is a 40-byte character array. Such an off-by-one overflow would not have been found without CRED instrumentation. The other three overflows are write overflows in a stack buffer that occur in the `sm` process during parsing of Jabber ID (JID). These overflows are triggered when the `user`, `host` or `resource` portions of the JID are larger than the `str` scratch buffer into which they are copied using `strcpy`.

It is worthwhile to note that the overflows were discovered using messages that consist of well-formed XML elements. While the protocol descriptions were designed to produce slightly malformed XML messages, all messages consisting of malformed XML were rejected by `jabberd`'s XML parser before any processing of their contents would take place. This shows that the Expat XML parser [9] used by `jabberd` is reasonably robust. It also demonstrates that messages that exhibit large deviations from the protocol will likely be dropped early in message parsing and will not reveal overflows hidden in further program paths.

The write overflows that occur during parsing of JID affect a buffer on the stack, and could potentially be exploited in a "stack-smashing" attack to run arbitrary code on the Jabber server. Alternatively, this vulnerability can be used to mount a denial of service

attack. The only mitigating factor is that the user is required to authenticate himself and start a session before being able to send these messages; however, many public servers allow any user to register.

Sending messages that cause overflows discovered by CRED to a gcc-compiled version of `jabberd` server produced some interesting results. The message that triggers the overflow at `jid.c: 115` did not cause a segfault in the uninstrumented version of `jabberd`, and thus would have been missed by the PROTOS system. While the messages that discovered the other two overflows triggered segfault behavior in the uninstrumented server, alternative messages could be constructed that produce the overflow but do not trigger segfault behavior. While all three overflows found in `jid.c` are detectable with an uninstrumented server, this test shows that they could be missed even if a message that produces the overflow is generated. A system that uses code instrumentation will recognize the overflows in such scenarios and will also find small overflows, such as the one in `main.c`, that do not generate a segfault.

Although not intended as a stress-test of the Jabber server, the grammar-based testing approach found several inconsistencies in `jabberd` internal structures that were not found when only a single message was sent to the server. Some of these inconsistencies manifested themselves in the server shutting down due to a failed `assert` statement inside `jabberd` code, while others just caused `jabberd` to exit mysteriously. Both of these mistakes enable a malicious user to perform a denial of service attack that causes the server to shutdown by sending a carefully-constructed sequence of messages. Though such an attack is not as dangerous as execution of malicious code on the system, it can nonetheless result in loss of revenue and productivity if the Jabber server is used in production systems that communicate via XML messages.

5.4 Discussion

Dynamic testing using code instrumentation and random messages has proved to be a useful technique in finding buffer overflows. Instrumenting the executable with fine-grained bounds checking enables much better buffer overflow detection than relying on segfaults. This instrumentation possesses an additional benefit of providing the developer with the information needed to isolate the problem. The automatic testing process is likewise an

advantage, since the developer need not create test cases or actively participate during the test.

However, there are some disadvantages to this approach as well. The protocol descriptions for random message generators had to be written by hand, since Jabber protocol description is described as an XML schema, which is not directly useful to either generator. The HMM description was particularly cumbersome to create, as it includes 59 states and 105 state transitions. Converting Jabber XML schema to the HMM description took about a week of effort. The HMM random message generator would be more useful for a protocol with shallow hierarchy or one where the order of message elements is particularly important for testing, as an HMM description supports fine-grained control of transitions between elements. Porting the Jabber protocol description to PCFG was significantly easier since it allows for deeply hierarchical structures, but still involved some challenges, as XML schemas allow for slightly different constructs than BNFs. The resulting grammar consists of 39 productions and took about two days to create. A PCFG is considerably less complex in structure than an HMM description for this protocol and can be easily created from BNF-type grammars that are frequently used to describe communications protocols.

Another shortcoming of the grammar-based dynamic testing approach is the lack of direction in exploring the input space. While random messages may provide a good representation of messages described by the protocol, the number of messages needed to cover the input space may be large. Since there is no feedback about which buffers are being affected by these messages, the bulk of the message corpus could be testing and finding the same overflow, instead of shifting attention to the less explored sections of the input space. Such direction and focus in the search would make the testing process much more efficient.

Chapter 6

Adaptive Test Case Synthesis

As demonstrated in Chapter 5, using code instrumentation in dynamic testing with random messages makes the process more efficient; however, the resulting approach still suffers from lack of direction in exploring the input space. Adaptive test case synthesis aims to mitigate this problem by using code instrumentation to provide feedback about test case effectiveness to the test case generator. With appropriate modifications, CRED can provide information about all buffer accesses that occur at runtime, and this knowledge can be used by the tester in constructing the next test case to drive the buffer closer to potential overflow.

The adaptive testing system presented in this chapter also explores a different method of generating test cases that relies on a database of captured traffic and techniques similar to data perturbation [40]. Simple static analysis of the source code is used to extract possible delimiting characters, which are used by the tester to parse the message into tokens that can be mutated automatically, without developer's involvement. These improvements result in a system that exhibits a more focused search over the inputs, uses fewer messages to find buffer overflows and requires less developer effort to use.

6.1 Dynamic Testing with Feedback

The overall architecture of a system utilizing adaptive test case synthesis for testing servers is shown in Figure 6-1. The bold filled arrows indicate the feedback loop in testing — messages are sent from the tester to the server, code instrumentation built into the server generates buffer access statistics, which are collected, stored in the database, and made available to the tester for analysis. The tester uses a database of messages that consists of

traffic captured at a deployed server. Alternatively, this message database can be populated by randomly generated messages. Static analysis module provides information needed to parse messages into tokens by finding *control characters* — character literals that appear in source code and are likely to delimit fields in the input. A modified CRED compiler (CRED-HW), discussed in Section 6.4.1, produces an instrumented executable that allows for collection of buffer access statistics, which are used to guide the tester in selecting further inputs.

The adaptive testing system uses the following algorithm:

1. **Select a message from the database.**
2. **Send the message to the server.**
3. **Record the *buffer access signature* for unaltered message.** This step creates the control *buffer access signature* — some subset or function of information about buffer accesses.
4. **Parse message into tokens.** This step utilizes *control characters* obtained by static analysis, as described in Section 6.5.
5. **For each token:**
   (a) **Mutate token.** Some possible mutations are described in Section 6.3.2.
   (b) **Reassemble the test message and send it to the server.**
   (c) **Record the buffer access signature for the test message.**
   (d) **Compare signatures of the original and test messages.** Find *targetable buffers* — buffers that are affected by the change introduced into the test message. Some metrics for finding such buffers are described in Section 6.4.2.
   (e) **If such buffers exist, then for all such buffers:**
      i. **Create a token that can potentially overflow a target buffer.** Creating such tokens is discussed in Section 6.4.2.
      ii. **Send resulting message to the server.** CRED will report a buffer overflow if one occurs.

   Otherwise, move on to the next token.

6.2 Message Database

The test case generator uses message mutation to explore the input space, and thus requires a corpus of messages. There are several different ways of obtaining such a message database, depending on resources and information available to the developer. Captured traffic can be used as a basis for mutation by the test case generator. Alternatively, random messages can

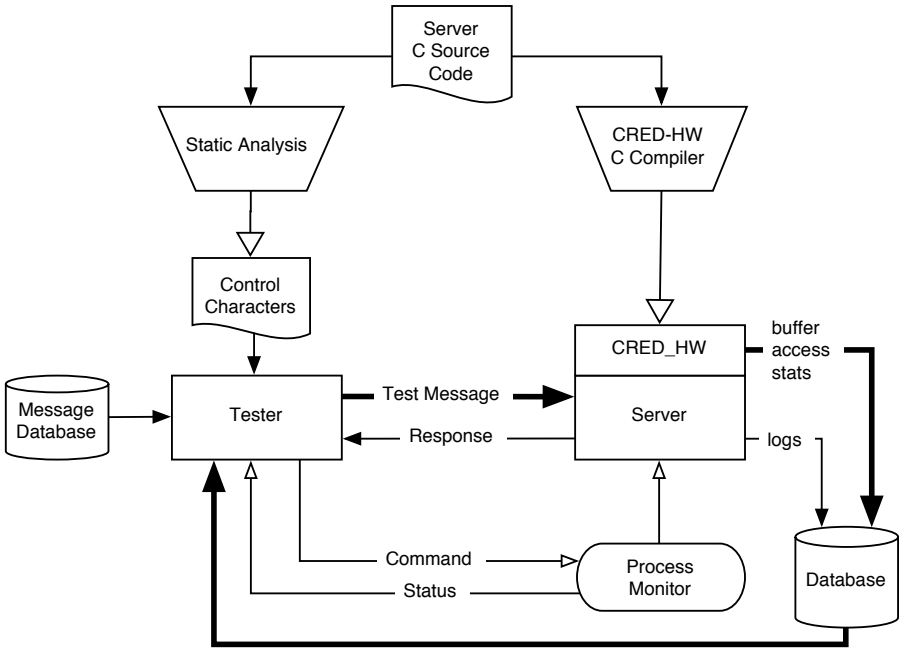

Figure 6-1: System architecture for adaptive testing. *Thick solid lines show the feedback loop. Control flow is shown by lines with unfilled arrowheads.*

be automatically generated if some description of the protocol is provided. The approaches are not entirely equivalent, however, as each plays a role in determining the way the input space is explored.

Collecting the messages at a deployed server is a simple way to populate the message database. It requires the least knowledge and effort, since creating a description of the protocol is not necessary. Captured traffic is also a good representation of messages the server is likely to encounter when deployed. However, mutating such messages without knowing the protocol will not explore code paths for message types not present in the trace. This can be a serious disadvantage if the typical traffic includes only several of many message types allowed under the protocol, as most problems lurk in rarely-used cases.

Another method of obtaining a message database is to use one of the random message generators, such as those described in the previous chapter. However, using these generators requires a developer to create the appropriate protocol description, which is not needed for captured traffic. The resulting messages will likely be semantically incorrect, and may even be syntactically incorrect, depending on the complexity level of protocol and the corresponding description. Using a random generator may be helpful, though, if a traffic capture

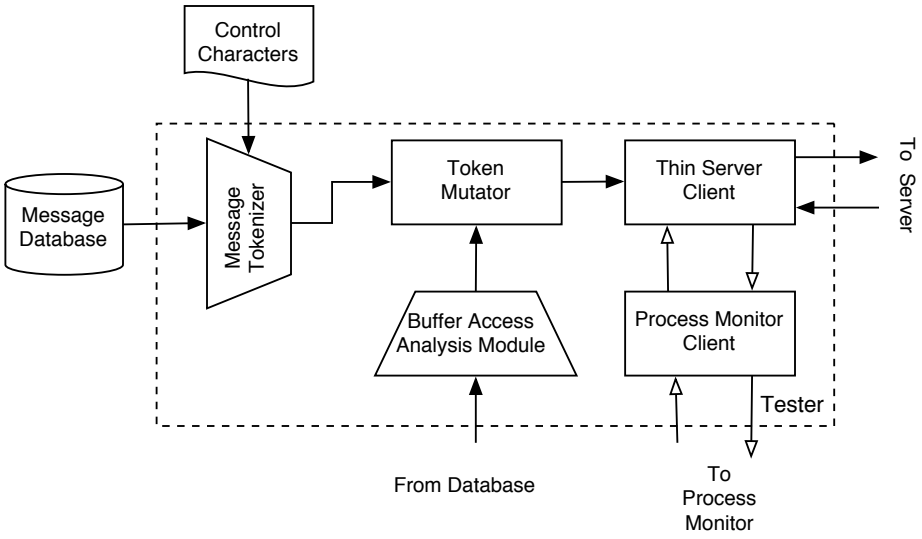

Figure 6-2: Test case generator architecture. *Data flow is shown by lines with black filled arrowheads. Control flow is shown by lines with unfilled arrowheads.*

only reveals a limited number of different message types. In the interests of simplicity, captured traffic is used to populate the message database during the evaluation of the adaptive testing system, as described in Section 6.6.

6.3 Test Case Generator

The test case generator works on a principle that is similar to *data perturbation* as described by Offutt in [40]. However, the test case generator presented here is both simpler and more flexible in what it tests — the message is split into tokens using control characters as delimiters. Thus, no structure is imposed on the resulting messages unlike Offutt's method, where created messages consist of well-formed XML. While the approach presented here results in less "intelligent" testing, it provides a much more simple and flexible system. Treating every token as a string allows the test case generator to explore type confusion in the server under test, though testing extreme limits of different types is not possible without knowing the protocol description.

Figure 6-2 presents the architecture of the test case generator. The test case generator utilizes *control characters* obtained from static analysis (as described in Section 6.5) in addition to feedback information from code instrumentation to mutate sample messages into test messages that target different buffers in the program. During a test run, a message

is selected from the database and passed through the *message tokenizer* that parses the message into tokens using control characters as delimiters. The resulting tokens are mutated inside the *token mutator* to produce test messages that are sent to the server under test. The feedback loop provides information about buffer accesses, which is processed inside the *buffer access analysis module*. The differences observed in buffer access signatures between the original and the mutated message are used by the token mutator for further input generation. The *message tokenizer* and *token mutator* modules are discussed in greater detail in this section, while the *buffer access analysis module* is described in Section 6.4.2.

Additional facilities are used by the tester to communicate with the server under test and to shutdown or restart the server. The *thin server client* communicates with the server by emulating a client that understands the server's protocol. This may be necessary if a session needs to be established before test messages can be sent. The *process monitor client* is used to send commands to the process monitor to inquire about server status or to restart or shutdown the server.

6.3.1 Tokenizing Messages

Tokenizing messages provides more fine-grained control over which buffer is targeted. If mutations were made to the message as a whole, it would likely be very difficult for the tester to understand how particular mutations affect different buffers. By tokenizing the message, the problem is reduced to determining changes in the buffer access signature due to changes in a particular token. This technique is especially useful when the tester can parse the message into the same tokens as the program under test; however, such an approach is also useful if the parsing is approximately correct.

If the protocol grammar is known, then the tester can parse the message according to the grammar. The advantage of such parsing is that the tester would be able to perform mutations that are semantically intelligent (on some level). The disadvantage of this approach is that parsing messages according to a grammar is rather difficult; in addition, it requires the developer to write a more complicated test case generator. Another downside is that violations of the protocol syntax will not be explored, as such parsing enforces a strict format on the message.

A simpler approach to tokenizing messages is simply to use the control characters obtained through static analysis as delimiters of the input message. While parsing in this

manner is hardly semantically intelligent, it is likely to produce syntactically correct interpretation of the message, which still allows intelligent mutations of the input. No knowledge of the protocol grammar is needed for this kind of parsing, and it is flexible enough to allow creation of messages that may violate the protocol syntax, thus testing this aspect of message parsing on the server as well. The main disadvantage of this approach is that the tester has no knowledge of semantic significance of message parts. Since this approach is simpler and requires less developer effort to use, it has been selected as the parsing method during evaluation of the adaptive testing system, described in Section 6.6.

6.3.2 Mutating Tokens

In the *data perturbation* model described by Offutt in [40], each token was recognized as belonging to some type, such as integer, string, float, etc. The mutations then utilized the knowledge of the type to change the token into extreme values — the integers were changed to `MAX_INT` and strings of maximum allowable length were generated. Since the parsing used by the test case generator does not determine token types, all tokens are treated as strings. Thus, the test case generator can only alter two things — token composition and token length. In the interests of a proof-of-concept system, only changes in token length have been explored; token composition was kept the same.

There are two separate functions of token mutation. During the first test run, the goal is to alter the original message slightly and observe corresponding changes in the buffer access signature. Perturbing the input in this manner allows the tester to determine which buffers can be targeted by making changes to a particular token. The other function of token mutation is to create overflow test cases. Once the targetable buffers are known, the information obtained from code instrumentation is used to create a test case that can potentially overflow the target buffer. Because the overflow test cases are tailored specifically to each target buffer, changing the length of each token to some maximum allowable value, as done in [40], is not directly useful in this kind of testing.

Other simple token mutations can also be used to test the robustness of the server. For example, a token may be removed from the message or the control characters enclosing the token could be swapped. While the proof-of-concept system does not perform these mutations, they can be easily applied by populating the message database with appropriately mutated messages, and then using the existing adaptive testing system.

6.4 Feedback Loop

The feedback loop is an integral part of the adaptive testing system. It is used to make the information obtained during test case execution available to the test case generator. Source code for the server is compiled with a modified C compiler to produce an executable with instrumentation that keeps track of information about buffer accesses. This information is analyzed and used by the test case generator to decide what input mutation should occur next. Searching the input space in this manner is more efficient and faster than sending random messages to the server and relying on segfaults to indicate an overflow, a technique used by the PROTOS project [28]. This section describes the buffer access information provided by the instrumentation and its analysis.

6.4.1 CRED-HW Instrumentation

Source code instrumentation is necessary to gather additional information about buffer accesses within the program. It is likely that any single input will not cause an overflow, and thus provides the test case generator with no additional information about what kinds of inputs to generate. However, adding instrumentation that shows the depth of access for each buffer provides the necessary information to correlate certain properties of the inputs to buffer access depth. Such instrumentation should not change the execution flow of the program, though it will incur additional overhead.

As determined in the evaluation of dynamic buffer overflow detection tools presented in Chapter 4, CRED is well-suited to provide the desired information. The bounds-checking code in CRED has been extended to keep track of *low water mark* and *high water mark* information about all buffers, that is, the minimum and maximum accesses to a buffer from its creation until it goes out of scope. These modifications were performed by Tim Leek and Kendra Kratkiewicz for a related project. The modified version of CRED is referred to as CRED-HW (for High Water marks) throughout the remaining discussion.

CRED-HW has two modes of operation — *trace* and *summary*. Under *trace* mode, CRED-HW prints out a line for each buffer access providing identifying information for the buffer, the depth of the access, and a timestamp whenever an access to a buffer occurs. Using this information, a complete timeseries of accesses for any buffer can be constructed. In the *summary* mode of operation, CRED-HW keeps track of buffer accesses, but only

Field	Description
name	The name of the object
file	The file where the object is created
line	The line where the object is created
size	Total size of the object in bytes
elt_size	The size of elements in bytes
num_elts	Number of elements in this object
min	The minimum index accessed in this object
max	The maximum index accessed in this object
storage	Memory location of the object (i.e. stack, heap or bss)
mem_addr	The object's address in memory

Table 6.1: Buffer access info fields provided by CRED-HW.

prints out a summary when the object is destroyed or goes out of scope. The information about each buffer available in *summary* mode is presented in Table 6.1.

This buffer access information is logged during program execution and then transfered into a database for easy access and analysis. Since running a large program or executing a test for a long time generates large quantities of buffer access information, CRED-HW has been created with the ability to log either to a file or to a named pipe. In the latter case, another process can be setup to read from the named pipe, analyze data as necessary and discard it, so that large quantities of storage are not needed.

6.4.2 Buffer Access Signature Analysis

Buffer access analysis module in the tester makes use of the information provided by CRED-HW and works to aggregate it for easier analysis. The raw data is examined, and a table is constructed for each object, which is uniquely identified by name, file, and line. Table 6.2 shows the fields present in this table.

The aggregation occurs inside a Perl script that reads a log file generated by CRED-HW. The stats are then loaded into a MySQL database for further analysis. Three tables are set up in MySQL — the *objects* table describes different buffers or objects encountered by the program, the *stats* table describes the access statistics for each object in each test run, and *affected_objects* lists the objects affected by a particular token (along with the token that affects them). The tester then uses this information to create test cases that will potentially overflow affected objects. Complete fields in each table are listed in Tables 6.3, 6.4, and 6.5.

Field	Description
name	The name of the object
file	The file where the object is created
line	The line where the object is created
size_low	The lowest (over all lives) total size of the object in bytes
size_high	The highest (over all lives) total size of the object in bytes
elt_size	The size of elements in bytes
num_elts	Number of elements in this object
lives	Number of times the object was created
total_accesses	Total number of accesses to this object
min_low	The lowest (over all lives) minimum access to this object
min_high	The highest (over all lives) minimum access to this object
max_low	The lowest (over all lives) maximum access to this object
max_high	The highest (over all lives) maximum access to this object
storage	Memory location of the object (i.e. stack, heap or bss)
mem_addr	The object's address in memory

Table 6.2: Aggregate buffer statistics fields. *This table is created from buffer access information provided by CRED-HW.*

An object is considered to be affected by a mutation if some statistic or a function of statistics for this object changes when some token is mutated. The *affected_objects* table is populated with objects and corresponding tokens that affect their statistics during the analysis phase. Different metrics could be used to determine which objects are affected; however, such metrics should be appropriate to the mutations that are performed by the tester. For the proof-of-concept system evaluated in this thesis, the chosen metric was the *max_high* statistic. Since the tester mutates messages by making them slightly longer, an increase in *max_high* statistic would indicate that the buffer is being accessed deeper than before and thus is affected by the mutation.

However, while this metric shows a correlation between token length and buffer access depth, it does not indicate whether the buffer could potentially be overflowed. For example, many buffers are dynamically allocated to the needed size. In this case, the *max_high* statistic will increase, but increasing the token size will not bring the buffer closer to overflow, as a larger buffer will be allocated. However, there are more interesting statistics, such as *size_high - max_high - 1*, which reflects the amount of unused memory in a buffer. If the value of this statistic decreases with mutation, then increasing the length of the token brings the buffer closer to overflow, so further mutation with appropriately-sized token should be attempted.

The size of the token to be used in the overflow message is determined by assuming a linear correlation between increase in *max_high* and the increase in token length. Nevertheless, the corresponding length increase may not be one-to-one, as some character representations use multiple bytes per character. Since *max_high* is measured in bytes and the token length is measured in characters, it is necessary to determine a conversion factor that can be used in the linear prediction of token length that will produce an overflow. The size of the overflow token (in characters) is calculated as follows:

$$len_{overflow} = \left\lfloor \frac{len_1 - len_0}{max\_high_1 - max\_high_0} \times (size\_high_1 - max\_high_1 - 1) \right\rfloor + len_1 + 1$$

In the formula above, *len* refers to length of the token in characters, *max_high* is the maximum access offset in bytes, and *size_high* is the largest size of the object in bytes. The variables with subscript 0 refer to the original message, whereas ones with subscript 1 refer to the mutated test message. The fraction $\frac{len_1 - len_0}{max\_high_1 - max\_high_0}$ represents the $\frac{char}{byte}$ conversion factor, which is likely not an integer. The $(size\_high_1 - max\_high_1 - 1)$ factor represents the number of unused bytes in the buffer — since max_high_1 is an index (in bytes), the adjustment of -1 is necessary. The conversion is `floor`ed to ensure that the resulting overflow is the smallest possible, and one extra character is added at the end to produce this overflow. The token with resulting length of $len_{overflow}$ is then used to test whether the buffer can be exploited with an off-by-one byte (or, if a single character is stored in x bytes, off-by-x bytes) overflow.

6.5 Static Analysis Module

Many mistakes resulting in buffer overflows occur when parsing an input stream. Different parts of the input stream are delineated by special character literals that separate different fields or values. Often, such characters are used in conditionals or loops within the program source or otherwise determine, directly or indirectly, the control flow of the program. Static analysis module in the adaptive testing system is used to find these *control characters* — character literals that appear in conditionals or loops within the source code. The static analysis module employed in the adaptive testing system uses C Intermediate Language (CIL) to parse the source code and perform analysis on appropriate parts. The process is described in more detail below.

Field	Description
object_id	Unique id for this object
process_name	Name of the process which uses this object
object_name	Name of the object
file	File where the object is created
line	Line where the object is created
storage	Memory storage type

Table 6.3: Fields in the `objects` table.

Field	Description
object_id	Unique id for this object
testrun	The number of the testrun
size_low	The smallest size (over all lives) in bytes
size_high	The largest size (over all lives) in bytes
lives	Number of lives
min_low	The lowest minimum access (over all lives) as an offset
min_high	The highest minimum access (over all lives) as an offset
max_low	The lowest maximum access (over all lives) as an offset
max_high	The highest maximum access (over all lives) as an offset

Table 6.4: Fields in the `stats` table

Field	Description
object_id	Unique id for this object
testrun	The number of the testrun
token	The string token affecting this buffer
token_index	The index of the token in the original message

Table 6.5: Fields in the `affected_objects` table.

6.5.1 CIL: C Intermediate Language

C Intermediate Language (CIL) [34] is a language for analysis and transformation of C programs. CIL offers a source-to-source transformation of C programs by parsing the C syntax and breaking down certain complicated C constructs into simpler ones. In CIL, all looping constructs are reduced to a single form, syntactic sugar such as "`->`" is eliminated, type declarations are separated from code, and scopes within function bodies are flattened, with appropriate alpha-renaming of variables. By converting source code to this intermediate representation, CIL disambiguates C syntax, such that memory accesses through pointers are represented differently than those via offsets.

CIL syntax has three basic concepts: an *expression*, an *instruction* and a *statement*. An *expression* represents functional computation without side effects or control flow. An *instruction* represents side effects, including assignments, function calls and embedded assembly instructions, but no local (intraprocedural) control flow. Finally, a *statement* is used to capture control flow of the program. In addition to these constructs, CIL contains other forms of representation for type declarations. Transforming C code into this simpler language makes many kinds of static analysis easier, since there are fewer constructs which present a more unified view of the program.

6.5.2 Source Analysis with CIL

In addition to parsing C code, CIL provides hooks for programmer-defined modules to perform analysis on the parsed source code and transform it if necessary. Such a module could be used to locate character literals embedded in conditional statements or otherwise involved in changing control flow of the program. Performing CIL analysis and modifications using the existing build process is made simple by a Perl script called `cilly` that can be used as a drop-in replacement for gcc. In order to perform CIL analysis, all invocations of `gcc`, `ar` and `ld` inside the makefile are changed to `cilly`, which emulates the appropriate utility in the make process. As the source code is parsed into CIL, `cilly` invokes registered user-defined modules and allows them to traverse the code tree, performing arbitrary analysis or modifications. Once the execution of modules is complete, the code tree is translated back into C source and passed to the appropriate program for further processing.

Tim Leek created a CIL module to extract literals from C source code for a related project and has kindly agreed to allow its use in the research presented here. This CIL module examines statements that change control flow, such as `if`-statements and loops, looking for literals embedded in the code. Literals that are found by the module are split into several categories — integer literals, float literals, character literals and string literals. Each group is then printed sorted by the number of occurrences of each literal in the source code. For the purposes of message parsing character literals are of most interest, as they are likely to delimit fields within the input message, thus controlling how far different buffers are filled.

6.5.3 Literals in `jabberd`

Section 6.6 describes an evaluation of the adaptive testing system using the Jabber Instant Messaging server. The static analysis module was used as part of the `jabberd` build environment to collect character literals that appear in the source code. The most commonly occurring character literals are presented in Table 6.6, along with the corresponding number of occurrences in source code. The complete list of 46 character literals found in `jabberd` source can be found in Appendix F. It is interesting to note that all the literals from Table 5.1, which detailed different field delimiters in Jabber messages, are present in Table 6.6. It is therefore likely that message parsing that occurs within the test case generator will be very similar to the one occurring inside `jabberd`, as desired for adaptive testing.

6.6 Evaluating Adaptive Testing System

In order to compare the adaptive testing system to the grammar-based dynamic testing approach presented in Chapter 5, the new system is also evaluated using the Jabber Instant Messaging server, described in Section 5.3.1. The goal of this evaluation is to determine whether the feedback loop improves the behavior observed in the evaluation of grammar-based testing with code instrumentation.

6.6.1 Test Environment

The test environment for the adaptive testing system is setup as shown in Figure 6-1. Test messages are generated by mutating sample captured traffic messages from the database.

Literal	Number of Occurrences
\000	54
=	5
|	4
space	4
/	3
\n	3
<	3
>	3
)	3
'	3
,	2
?	2
&	2
(	2
:	2
@	2

Table 6.6: Literals found in `jabberd`. *A complete listing is included in Appendix F*

CRED-HW provides code instrumentation that records buffer access statistics for each buffer in addition to reporting any access violations. As before, a process monitor is used to restart `jabberd` when necessary. During tests `jabberd` is treated as a single entity, and all processes, including the tester and all `jabberd` processes run on the same computer. The machine used in this testing is a dual 2.66GHz Xeon CPU desktop, running Red Hat Linux release 9 (Shrike).

The major difference in the testing environment from the evaluation presented in the previous chapter is the use of a database to store CRED-HW buffer access statistics. This addition is necessary to make the information easily available to the tester module for analysis. Since `jabberd` uses a MySQL server to store authentication and contacts information, it was convenient to use the same server to store CRED-HW buffer access information.

6.6.2 Test Procedure

The Jabber server was compiled with CRED-HW compiler, and the executable was tested using the framework shown in Figure 6-1. A copy of the source code for `jabberd` was passed through the static analysis module to extract control characters that are presented in Appendix F. A file containing these characters was provided to the test case generator, thus enabling it to parse messages into tokens by using control characters as delimiters. The adaptive testing system then followed the steps outlined in Section 6.1 to mutate all non-

```
<message
  to='alice@swordfish.ll.mit.edu/resource'
  from='bob@swordfish.ll.mit.edu/test'
  id='12345'>
    <thread>
      thread1 thread2 thread3
    </thread>
    <subject>
      subject1 subject2 subject3
    </subject>
    <body>
      body1 body2 body3
    </body>
</message>
```

Figure 6-3: A *legal* Jabber IM message.

delimiter tokens, determine affected buffers and attempt to overflow them with a mutated message.

Token mutations consisted purely of length mutations, while keeping token composition the same. This was achieved by creating a histogram of characters in the original token and using this histogram as a probability distribution to pick new characters when an increase in length was needed. After recording a control buffer access signature by sending the original message to the server, a test message was generated by increasing the length of the token under test by 10 characters. The new test message was then sent to the server to determine affected objects, for which potential overflow messages were then constructed according to the formula in Section 6.4.2.

The message database consisted of a two sample messages: a *legal* message, shown in Figure 6-3 and a *malformed* message, shown in Figure 6-4. The legal message adheres to the protocol syntactically and semantically — it was obtained from captured traffic on a local Jabber server. The malformed message was created by hand from the legal message; however, it could easily have been generated by a random message generator with an appropriate protocol description.

Mutating tokens of the legal message explores the common path of messages through the server and tests whether the bounds of all buffers containing message contents have been correctly calculated. Mutating tokens of the malformed message tests type confusion in the server. Since the `'@'` and `'/'` control characters in the recipient's JID are reordered, parsing the JID may proceed incorrectly. Performing length mutations on tokens from this

```
<message
  to='/alice@swordfish.ll.mit.eduresource'
  from='bob@swordfish.ll.mit.edu/test'
  id='12345'>
    <thread>
      thread1 thread2 thread3
    </thread>
    <subject>
      subject1 subject2 subject3
    </subject>
    <body>
      body1 body2 body3
    </body>
</message>
```

Figure 6-4: A *malformed* Jabber IM message. *The '@' and '/' delimiters have been misordered in the recipient's JID.*

malformed message may create accesses to different buffers or through different code paths that could be vulnerable. Both tests are designed to find small overflows.

Another distinction from the evaluation in Chapter 5 is that `jabberd` was restarted after each message. The restarts were necessary to ensure that the initial state of the server was the same for each test run, since a buffer access pattern was observed in order to discern the effect of a mutation. As in the previous evaluation, the messages causing the overflow were sent to a gcc-compiled version of `jabberd` to determine whether such a message would be detectable without CRED's instrumentation.

6.6.3 Sample Test Run

This section presents a sample test run of the adaptive testing system for a particular token of the *legal* message, shown in Figure 6-3. First, the original message is sent to the server and the corresponding buffer access signature is recorded in the database. Figure 6-5 shows the desired metric ($size\_high - max\_high - 1$) for all 789 objects that were accessed during processing of the original message. Bars for most dynamically allocated objects have the height of zero on this graph, as generally the size of the data is used to determine the amount of memory that is allocated; thus, the entire buffer is accessed.

The sample test run focuses on changes to the `host` portion of the recipient's JID. The original value is ``swordfish.ll.mit.edu''; the tester increases its length by 10 characters while keeping the token composition the same, thus replacing the original value with ``swordfish.ll.mit.edudfw..i.wdo''. The test message is then sent to the server and

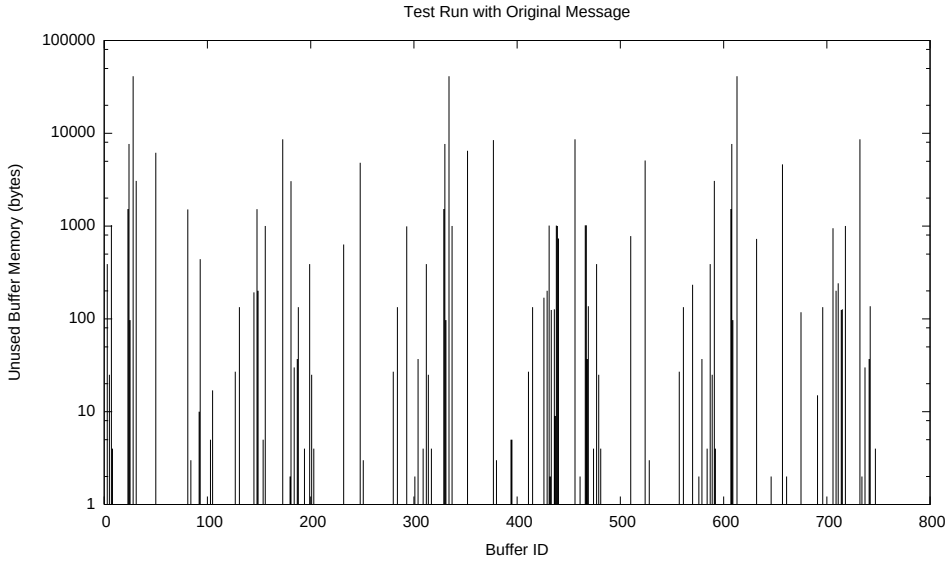

Figure 6-5: Buffer access signature for the original message. *This serves as a control signature to which buffer access signatures generated by test messages are compared.*

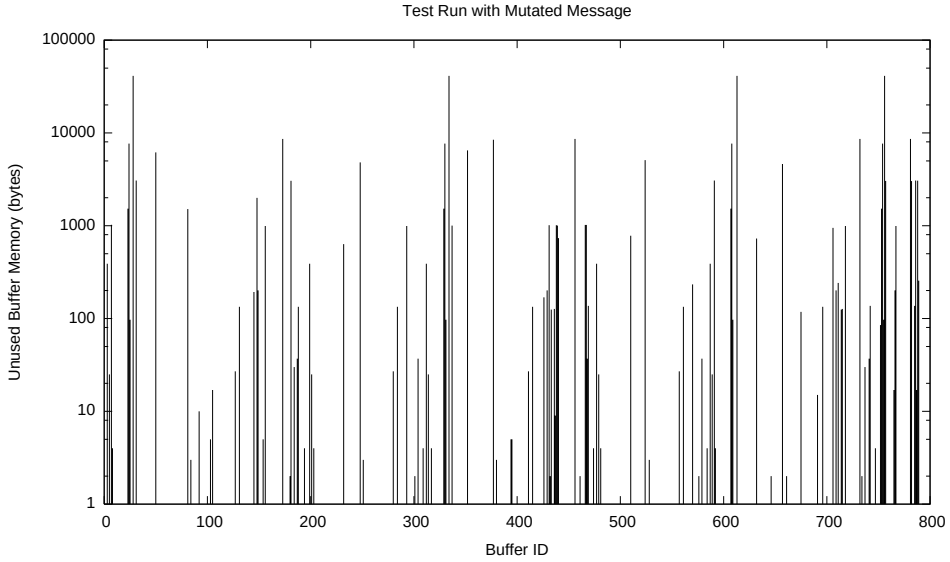

Figure 6-6: Buffer access signature for a mutated message. *For this mutated message, the host part of the recipient's JID is increased by 10 characters.*

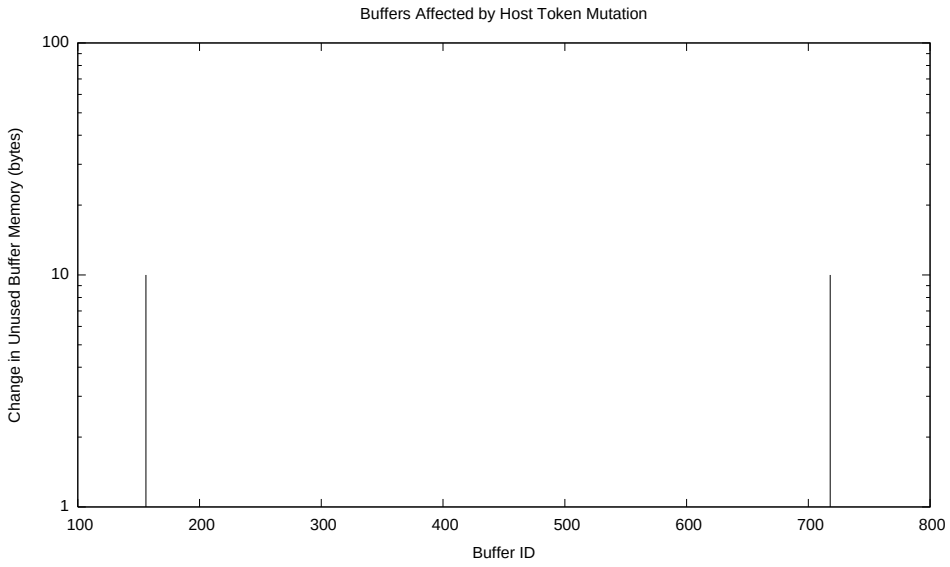

Figure 6-7: Difference in buffer access signatures between the original and mutated messages.

Object Name	Process Name	File	Line	Storage
malloc	sm	jid.c	175	heap
malloc	sm	nfkc.c	360	heap
malloc	sm	nfkc.c	471	heap
malloc	sm	nfkc.c	778	heap
realloc	sm	stringprep.c	375	heap
str	sm	jid.c	81	stack
malloc	router	jid.c	175	heap
malloc	router	nfkc.c	360	heap
malloc	router	nfkc.c	471	heap
malloc	router	nfkc.c	778	heap
realloc	router	stringprep.c	375	heap
str	router	jid.c	81	stack

Table 6.7: Buffers with changed `max_high` statistic. *This table shows all buffers for which* ***`max_high`*** *statistic changed between the original and mutated message test runs.*

Object Name	Process Name	File	Line	Storage
`str`	`sm`	jid.c	81	stack
`str`	`router`	jid.c	81	stack

Table 6.8: Targetable buffers. *For these buffers the value of* `size_high` - `max_high` *decreased due to the token mutation.*

the corresponding buffer access signature is collected. The resulting buffer access signature is shown in Figure 6-6. When the test signature is compared to the original, the buffers shown in Table 6.8 are determined as targetable. As a comparison, Table 6.7 lists all buffers for which *max_high* is larger in the test signature than in the original signature. It is worth noting that most of those buffers are dynamically allocated and completely filled; thus, targeting these buffers is likely to be fruitless.

The target buffers belong to the same piece of utility code that parses the JID; however, the buffers exist in different processes, so they are considered different objects. The remaining space in each buffer is 994, which is 10 bytes smaller than during the original test run; thus, each character is recorded as one byte. This could be different, especially if the characters are converted to another character set, such as Unicode. The total length of the token is computed to be 1025 characters, which requires generation of additional 995 characters to concatenate to the existing hostname.

The potential overflow message is generated for each buffer and sent to the `jabberd` server. Both messages find an overflow in `jid.c, line 115` (see Appendix C); however, both times the overflow is observed in the `sm` process, even though one of the target buffers is in the `router` process. This is due to the data flow through the `jabberd` server. The `sm` process receives the message first and parses it; since the `host` section of the JID is too large, there is a buffer overflow and CRED shuts the server down. However, if no overflow exists, the message is forwarded to the `router`. Thus, both buffers show up as being targetable, but only one of the overflows can actually be exploited.

6.6.4 Test Results

The overflows found by adaptive testing using a *legal* message are shown in Table 6.9, while the ones found using a *malformed* message are shown in Table 6.10. The tables list the process and line of code where the access violation occurred, the name of the object, location of its creation and its storage type. The *message count* column shows the number of times

Process	Access Violation	Object	Created at	Storage	Type	Message Count	Detected by Segfault
s2s	main.c: 129	secret	main.c: 65	stack	write	1	No
sm	jid.c: 103	str	jid.c: 81	stack	write	1	No
sm	jid.c: 115	str	jid.c: 81	stack	write	3	No
sm	jid.c: 127	str	jid.c: 81	stack	write	2	No

Table 6.9: Overflows detected using adaptive testing with a *legal* message. *Message count shows the number of times the overflow was found during the 53 message test run. The overflow is "detected by segfault" if a version of* `jabberd` *compiled with gcc segfaults on the message that causes this overflow.*

Process	Access Violation	Object	Created at	Storage	Type	Message Count	Detected by Segfault
s2s	main.c: 129	secret	main.c: 65	stack	write	1	No
router	jid.c: 103	str	jid.c: 81	stack	write	2	No
sm	jid.c: 127	str	jid.c: 81	stack	write	3	No
sm	jid.c: 173	realloc	nad.c: 79	heap	read	14	No

Table 6.10: Overflows detected using adaptive testing with *malformed* message. *Message count shows the number of times the overflow was found during the 39 message test run. The overflow is "detected by segfault" if a version of* `jabberd` *compiled with gcc segfaults on the message that causes this overflow.*

this overflow was detected during the test. The actual error messages for these overflows are presented in Appendix C, and the corresponding overflow-causing messages can be found in Appendix E.

Using the adaptive testing system with a *legal* message reveals the same overflows that were found during dynamic testing with code instrumentation, described in Chapter 5. A discussion of these overflows can be found in Section 5.3.4. There are, however, some differences in the results that are due to employing a feedback loop and adaptive test case synthesis. The duration of the test was only 6.5 minutes, which is 5.3 times shorter than the length of the grammar-based test. This speed increase was possible because the entire test consisted of sending only 53 messages, much fewer than 10,000 messages used in the random message test. Since token mutation was explored in a systematic manner, there was little overlap — only one message was created to test each overflow. Since JID parsing occurs twice for each message (once for the sender and again for the recipient), these overflows have been found more than once. However, it is possible to keep a cache of buffers that have

been already explored and thus only generate one message for each potential buffer overflow. Such an extension for random message testing could not be done easily, as buffers affected by each message are not known, and processing of the same input is explored multiple times. This demonstrates that the adaptive testing system makes more efficient use of messages.

Another interesting feature of these results is that none of the messages constructed to overflow different buffers generated a segfault when sent to a gcc-compiled version of `jabberd`. This demonstrates that the technique used in adaptive testing is effective for finding small overflows that may be exploitable but would be missed by a dynamic testing system without code instrumentation.

Using the adaptive testing system with a *malformed* message reveals another overflow, that was not found during grammar-based random message testing. While the protocol descriptions were designed to generate such malformed messages, it is possible that some other mutation also occurred in this kind of message, and thus this overflow was not found. Since the adaptive testing approach uses a systematic method of mutating tokens, changes are considered one at a time, which helps to find errors in the source code. The new overflow in `jid.c, line 173` is a read overflow in the heap buffer. Another error found by the adaptive testing system with a *malformed* message is a `NULL` pointer dereference, that occurs during a `strcmp` in `jid.c, line 281`. Causing this error results in immediate denial of service attack, as a gcc-compiled version of `jabberd` will segfault. While these errors cannot be exploited for arbitrary code execution, they could be used to generate a denial of service attack on the server. Such attacks are still very costly for production systems and companies that rely on e-commerce.

Another difference between the adaptive testing results for a *legal* message and for a *malformed* message is the location of the `jid.c, 103` overflow. When using a *malformed* message, the access violation occurs within the `router` process, instead of the `sm` process. While this particular piece of source code is shared between the two processes, finding this overflow shows that a different code path has been explored by using a malformed message and generating test inputs in a systematic way.

6.7 Discussion

The evaluation results show that the adaptive test case generation approach discussed in this chapter improves on the results of dynamic testing with random messages. Using code instrumentation and a feedback loop provides direction for exploring the input space and results in much fewer messages necessary to find an overflow. Since mutations are explored systematically, using the information obtained from code instrumentation, the overflow messages are created to target particular buffers. The resulting messages can find even small overflows that would not be detected by dynamic testing without code instrumentation. The data perturbation approach makes the adaptive testing system easy to use, as no protocol description needs to be created to generate test messages. The use of static analysis automates finding character literals that are used to parse messages into tokens, thus minimizing developer involvement in the process. In addition, this approach to tokenizing messages allows for more flexible testing, since resulting test messages do not have to adhere strictly to the protocol.

The proof-of-concept implementation of the adaptive testing system described in this chapter considered only length increases of non-delimiter tokens as mutation tests. However, the adaptive test case generation approach is quite general, and any number of tests and metrics can be used to evaluate server response to different error conditions. Considering more complicated functions of buffer access statistics may yield a more complete understanding of program behavior and thus allow for more intelligent testing. In addition, the system could be used for a limited version of dynamic taint analysis by determining which buffers are affected by certain input mutations.

The adaptive testing system is, of course, not without some disadvantages. While using captured traffic in the message database is an attractive approach because of its simplicity, the messages are likely to represent only a subset of the protocol, and thus some code paths and buffers will remain untested, since no other protocol description is provided to the tester. The adaptive testing approach also requires that the environment remain the same between the test run with the original message and the test run with the mutated message, as only in that case will comparing buffer access signatures be meaningful. However, it may be quite difficult to ensure that the environment does not change — the program may perform some actions based on the output of a random generator, system time or response

from the DNS server. Controlling such external dependencies is very difficult and may require building a virtual environment for the program under test.

Another possible limitation of the adaptive testing system is speed. While a more effective search over the input space drastically reduces the number of messages needed to find the overflows, the adaptive testing system requires that the server is restarted after each message, which results in large overhead. While only 53 messages were sent during the test run of the adaptive testing system, the test took 6.5 minutes, which corresponds to a message rate of 8.15 messages per minute. During the grammar-based test run, however, messages were sent at a rate of 290 messages per minute. In addition to server restarts, the increase in overhead is also caused by the extra data processing and additional code instrumentation.

Further development and evaluation of this approach are necessary to build a testing system that will utilize the full potential of this method. Some limitations of the current system and suggestions for further development are discussed in Chapter 7. However, even in its proof-of-concept implementation, the ideas of code instrumentation and feedback improve on the results of the dynamic testing with random messages and demonstrate a valuable testing framework.

Chapter 7

Future Work

The adaptive testing system implementation described in Chapter 6 is a prototype that uses simple ideas for tokenizing messages, mutating tokens and analyzing buffer access information. Even with these simple ideas the system succeeded in finding buffer overflows in `jabberd` server. However, improvements can be made in all these areas to create a system that will more fully utilize the potential of adaptive test case synthesis. This chapter details some of the limitations of the existing implementation and suggests modifications to the adaptive testing system that may enable it to perform better in detecting buffer overflows.

7.1 Tokenizing Messages

The existing proof-of-concept adaptive system implementation splits the message into tokens by using character literals found in the source code as delimiters. While such approach is very simple to implement, it is likely that parsing the message in this way is inexact, and results in the tester misinterpreting token bounds. Such ad-hoc parsing also precludes the tester from learning about hierarchical constructs or other dependencies that are likely present in the message. If instead a protocol description were provided by the developer, the tester would be able to gain new insight into how the message is segmented into tokens inside the server, and thus would be able to produce better test cases. Alternative mutations, such as changing the scope of different elements could also be explored automatically if such knowledge were available to the tester.

7.2 Token Mutations

For simplicity, only length mutations were explored as part of the adaptive testing system presented in this thesis. However, more complex mutations may lead to a better exploration of the program's ability to handle different inputs and will likely find more overflows. A simple extension of the existing system could perform type recognition for each token, thus assigning types such as *int*, *float*, *date*, *time* or *string* to each token. Once such types are assigned, the perturbations of the token value would be based on its type, instead of treating each token as a string. This approach would allow for more intelligent mutations that may explore different code paths in the server.

Alternative tests could also be conducted to explore more complex relationships between inputs and buffer access statistics. For example, integer tokens could specify the length or the number of elements in the following token. This correlation would be reflected in a changing buffer size or different number of instantiations. Once such relationship is established, it could also be used to drive the affected buffer closer to overflow. With knowledge of the protocol description, even more complicated test cases utilizing semantic dependencies between message elements could be constructed automatically. This would result in more intelligent automatic testing and provide a more thorough test of server's robustness. More development and testing is needed to evaluate effectiveness and applicability of these approaches.

The adaptive testing system would also benefit from the use of a more systematic learning algorithm than the existing ad-hoc approach. For example, applying a gradient descent algorithm to the *size_high* - *max_high* metric may lead to discovery of more affected buffers and buffer overflows. Currently, the system uses a simple mutation to probe for affected buffers, which is only performed once. Likewise, only a single attempt is made at overflowing an affected buffer. Using a gradient descent algorithm would enable a more systematic exploration of the relationship between input tokens and affected buffers and may result in discovery of additional overflows.

7.3 Buffer Instances

The current implementation of the adaptive testing system uses only aggregate statistics about buffer accesses; however, aggregating data results in lost information that may enable

the system to perform better. As described in Section 6.4.1, CRED-HW is capable of providing detailed information about each buffer access. The problem lies in aggregating this information correctly — that is, in providing each buffer with a unique identity. For example, if a function is called multiple times, a local buffer will be instantiated multiple times. Under the existing aggregation scheme, buffer access information from all such "lives" of a buffer is collected together. This can present a problem — suppose that the same scratch buffer is used for a short token during one function call and a long token during another. The use of this buffer by the short token may be "masked" — the *min_low* and *max_high* statistics will correspond to the use of this buffer by the long token, provided that the difference in length is larger than the change in the short token due to test mutation. Worse yet, the aggregate buffer access statistics may be mixing together information from different lives of the buffer, thus presenting the tester with a distorted picture.

This problem appears particularly often in dynamically allocated buffers. Since dynamically allocated buffers usually grow in size with the size of the token, *max_high* and *size_high* reflect only statistics about the longest token that used this buffer, thus masking uses by other tokens. It would thus be more useful if buffer access statistics were collected separately for each buffer *instance*, that is, each instantiation of the buffer would be treated as a different object for the purposes of the analysis. The analysis module would be able to use this extra information to determine better correlation between tokens and buffers. However, the number of objects and size of the corresponding database tables would significantly increase, thus slowing down testing even further.

Chapter 8

Conclusions

This thesis explored the use of code instrumentation and feedback in test case synthesis for dynamic buffer overflow detection. Such methods have been shown to be more effective than existing approaches at directing the search through the input space, finding a revealing input and detecting the overflows when they occur. A proof-of-concept adaptive testing system has demonstrated that combining dynamic testing with information about buffer accesses obtained through code instrumentation can be used to build a comprehensive testing framework to detect buffer overflows. Important points discovered during the research involved in designing, implementing and evaluating this system are summarized in this chapter.

8.1 Required Instrumentation

Effectively searching through the input space would not be possible without feedback that provides information about buffer access patterns associated with a particular input. Such feedback, in turn, would not be possible without a fine-grained bounds-checking tool. As discussed in Section 4.6, fine-grained memory monitoring is key to detecting small buffer overflows, which can still be exploited to give the attacker control of the victim's system.

It is likewise essential that the tool providing instrumentation reports errors at the time of occurrence. This allows the tool to provide accurate information about the error that has occurred, and also preserves as much program state as possible. Reporting the error at the time of program termination may produce an incorrect result and direct the developer to the wrong location in the source code. In addition, altering program state in this manner can have very undesirable consequences.

Another important feature of a dynamic buffer overflow detection tool is ability to detect overflows in library functions. Since many overflows occur within different string or I/O functions provided by the C library, it is necessary to be able to detect these overflows. As discussed in Chapter 4, many tools take a wrapper approach to detecting overflows in library functions; however, writing wrappers is tedious and error-prone. A better solution would involve compiling the C library with the tool to enable buffer overflow detection within the library. However, this solution would result in a performance penalty, due to the overhead created by the instrumentation.

Perhaps the biggest hurdle observed during the evaluation of dynamic buffer overflow detection tools was compiling large complicated programs. Since it is nearly impossible to perform an effective code review of a large complex program, it is essential that an automatic technique, such as the adaptive testing approach or dynamic testing, is used to evaluate the program's robustness. This, in turn, requires that the dynamic buffer overflow detector used in such a system can compile complex programs without incurring a large performance overhead. As discussed in Chapter 4, CRED meets these requirements, and it has been extended for use in grammar-based dynamic testing and adaptive testing system implementations included in this project.

8.2 Generating Test Cases

While dynamic testing methods solve the problem of false alarms that render static analysis methods impractical, they present another problem — finding an input that will trigger the buffer overflow. Since the input space is generally very large, enumerating all possible inputs is rarely practical. One approach to exploring the input space is random message generation. This approach requires a protocol description, and two such random message generators were discussed in Chapter 5. With proper protocol descriptions, these generators can create test cases that probe different parts of the input space to test the robustness of the program. However, without feedback the resulting system lacks direction in its search over inputs and may find the same overflow multiple times. In addition, even with fine-grained bounds-checking instrumentation, the testing framework can only determine when an overflow has actually occurred, and may miss an overflow because its input test case was just a byte too short.

8.3 Advantages of the Adaptive Testing Approach

The adaptive testing approach offers a solution to the problems described above. It utilizes a feedback loop to direct the search through the input space, as information about buffer access patterns is made available to the test case generator. The adaptive testing system is also easier to use as it mutates messages collected by capturing traffic and thus does not require a developer to create a protocol description or generate test cases manually. Because of feedback, the adaptive testing system focuses on particular buffers and can design test cases that target particular parts of the program. In addition, the adaptive testing approach uses information from static analysis of the source code to provide some (albeit limited) understanding of the message, which allows for more intelligent parsing and token mutations. These features make the adaptive testing approach a more comprehensive testing system, which remains practical and easy to use. Once combined with improvements described in Chapter 7, the adaptive testing approach promises to become a flexible and comprehensive solution for automatic buffer overflow detection before deployment.

Appendix A

HMM Description

This appendix presents the HMM description used by the random message generator to create test messages for Jabber Instant Messaging server. As discussed in Chapter 5, the HMM description consists of the following elements:

state Each state is described by a `<state/>` XML element. A unique *name* is specified for each state as an attribute. Each state contains a multinomial probability distribution over *tokens*.

token Each token is described by a `<token/>` XML element. A *name*, that is unique within a state, is specified for each token as an attribute. The *charset* and *prob* attributes specify the composition of the token and its probability of occurrence. Tokens are produced by generator functions specified by the *generator* element.

generator The token generator function call is described by a `<generator/>` XML element. The appropriate arguments to the generator functions appear enclosed in `<arg/>` XML tags. The first argument to any generator is the name of the character set of the enclosing token. In the current implementation, the following generators exist:

- **constant** Accepts a string argument and returns it verbatim.
- **gen_constlen_str** Accepts a number specifying the length of the string to be generated. Characters are picked from the provided charset.
- **gen_varlen_str** Accepts a number specifying a stop probability. This generator produces strings with lengths that are distributed geometrically. Characters are picked from the provided charset.
- **gen_jid** Accepts a number specifying a stop probability. Generates three strings using *gen_varlen_str*, and returns `str1@str2/str3` as a randomly generated JID.

state-trans Each state transition is described by a `<state-trans/>` XML element. The source and destination states are specified as *from* and *to* attributes, respectively. The *prob* attribute specifies the probability of following this transition.

start-state Each starting state is specified by a `<start-state/>` XML element. It includes the *name* of the starting state and a corresponding *prob* attribute.

charset The character set is specified by a `<charset/>` XML element. It defines a discrete probability distribution over *symbols*. It includes a unique *name* attribute.

symbol A symbol is described by a `<symbol/>` XML element. It specifies a literal character and the probability with which this character should appear.

The probability attribute is optional; if it is not specified, all elements in the affected probability distribution are equally likely. The entities defined at the beginning of the HMM description provide a mechanism for mitigating some cases of redundancy. All references to `&ent-name;` are replaced with the contents of the entity with name `ent-name`; however, no parameterization is possible — the replacement XML appears verbatim at the beginning of the description. Some elements are separated by blank lines to increase readability.

```
<?xml version="1.0"?>
<!DOCTYPE test-config
[
  <!ENTITY open-ab-token
            "<token name='open-ab' charset='open-ab'>
               <generator func='gen_constlen_str'>
                 <arg>1</arg>
               </generator>
             </token>">

  <!ENTITY close-ab-token
            "<token name='close-ab' charset='close-ab'>
               <generator func='gen_constlen_str'>
                 <arg>1</arg>
               </generator>
             </token>">

  <!ENTITY quote-token
            "<token name='quote' charset='quote'>
               <generator func='gen_constlen_str'>
                 <arg>1</arg>
               </generator>
             </token>">

  <!ENTITY slash-token
            "<token name='slash' charset='slash'>
               <generator func='gen_constlen_str'>
                 <arg>1</arg>
               </generator>
             </token>">
]>

<test-config>
  <state-machine name='message'>

<!-- Empty message states -->
    <state name='emsg-open-ab'>&open-ab-token;</state>

    <state name='emsg-tag-name'>
      <token name='tag-name'>
        <generator func='constant'>
          <arg>message</arg>
        </generator>
      </token>
    </state>


    <state name='emsg-attrib-to'>
      <token name='to-valid' prob='0.9'>
        <generator func='constant'>
          <arg> to=&apos;</arg>
        </generator>
      </token>
      <token name='to-no-equals-quote' prob='0.1'>
```

```
      <generator func='constant'><arg> to</arg></generator>
    </token>
    <token name='to-no-quote' prob='0.1'>
      <generator func='constant'><arg> to=</arg></generator>
    </token>
    <token name='to-no-equals' prob='0.1'>
      <generator func='constant'>
        <arg> to&amp;apos;</arg>
      </generator>
    </token>
  </state>

  <state name='emsg-attrib-to-val'>
    <token name='jid'>
      <generator func='gen_jid'><arg>0.05</arg></generator>
    </token>
    <token name='alice'>
      <generator func='constant'>
        <arg>alice@swordfish.ll.mit.edu/test</arg>
      </generator>
    </token>
    <token name='bob'>
      <generator func='constant'>
        <arg>bob@swordfish.ll.mit.edu/test</arg>
      </generator>
    </token>
  </state>

  <state name='emsg-attrib-to-end-quote'>&amp;quote-token;</state>

  <state name='emsg-attrib-from'>
    <token name='from-valid' prob='0.9'>
      <generator func='constant'>
        <arg> from=&amp;apos;</arg>
      </generator>
    </token>
    <token name='from-no-equals-quote' prob='0.1'>
      <generator func='constant'><arg> from</arg></generator>
    </token>
    <token name='from-no-quote' prob='0.1'>
      <generator func='constant'><arg> from=</arg></generator>
    </token>
    <token name='from-no-equals' prob='0.1'>
      <generator func='constant'>
        <arg> from&amp;apos;</arg>
      </generator>
    </token>
  </state>

  <state name='emsg-attrib-from-val'>
    <token name='jid'>
      <generator func='gen_jid'><arg>0.05</arg></generator>
    </token>
    <token name='alice'>
```

```
        <generator func='constant'>
          <arg>alice@swordfish.ll.mit.edu/test</arg>
        </generator>
      </token>
      <token name='bob'>
        <generator func='constant'>
          <arg>bob@swordfish.ll.mit.edu/test</arg>
        </generator>
      </token>
    </state>

    <state name='emsg-attrib-from-end-quote'>&quote-token;</state>


    <state name='emsg-attrib-type'>
      <token name='type-valid' prob='0.9'>
        <generator func='constant'>
          <arg> type=&apos;</arg>
        </generator>
      </token>
      <token name='type-no-equals-quote' prob='0.1'>
        <generator func='constant'><arg> type</arg></generator>
      </token>
      <token name='type-no-quote' prob='0.1'>
        <generator func='constant'><arg> type=</arg></generator>
      </token>
      <token name='type-no-equals' prob='0.1'>
        <generator func='constant'>
          <arg> type&apos;</arg>
        </generator>
      </token>
    </state>

    <state name='emsg-attrib-type-val'>
      <token name='chat'>
        <generator func='constant'><arg>chat</arg></generator>
      </token>
      <token name='error'>
        <generator func='constant'><arg>error</arg></generator>
      </token>
      <token name='groupchat'>
        <generator func='constant'><arg>groupchat</arg></generator>
      </token>
      <token name='headline'>
        <generator func='constant'><arg>headline</arg></generator>
      </token>
      <token name='normal'>
        <generator func='constant'> <arg>normal</arg> </generator>
      </token>
    </state>

    <state name='emsg-attrib-type-end-quote'>&quote-token;</state>

    <state name='emsg-attrib-id'>
```

```
      <token name='id-valid' prob='0.9'>
        <generator func='constant'>
          <arg> id=&apos;</arg>
        </generator>
      </token>
      <token name='id-no-equals-quote' prob='0.1'>
        <generator func='constant'><arg> id</arg></generator>
      </token>
      <token name='id-no-quote' prob='0.1'>
        <generator func='constant'><arg> id=</arg></generator>
      </token>
      <token name='id-no-equals' prob='0.1'>
        <generator func='constant'>
          <arg> id&apos;</arg>
        </generator>
      </token>
    </state>

    <state name='emsg-attrib-id-val'>
      <token name='id' charset='numeric'>
        <generator func='gen_varlen_str'>
          <arg>0.05</arg>
        </generator>
      </token>
    </state>

    <state name='emsg-attrib-id-end-quote'>&quote-token;</state>

    <state name='emsg-close-slash'>&slash-token;</state>

    <state name='emsg-close-ab'>&close-ab-token;</state>

  <!-- Empty message paths -->
    <state-trans from='emsg-attrib-to' to='emsg-attrib-to-val'/>
    <state-trans from='emsg-attrib-to-val'
                 to='emsg-attrib-to-end-quote'/>

    <state-trans from='emsg-attrib-from'
                 to='emsg-attrib-from-val'/>
    <state-trans from='emsg-attrib-from-val'
                 to='emsg-attrib-from-end-quote'/>

    <state-trans from='emsg-attrib-type'
                 to='emsg-attrib-type-val'/>
    <state-trans from='emsg-attrib-type-val'
                 to='emsg-attrib-type-end-quote'/>

    <state-trans from='emsg-attrib-id' to='emsg-attrib-id-val'/>
    <state-trans from='emsg-attrib-id-val'
                 to='emsg-attrib-id-end-quote'/>

    <state-trans from='emsg-open-ab' to='emsg-tag-name'/>
    <state-trans from='emsg-tag-name'
                 to='emsg-attrib-to' prob='0.24'/>
```

```
        <state-trans from='emsg-tag-name'
                     to='emsg-attrib-from' prob='0.24'/>
        <state-trans from='emsg-tag-name'
220                  to='emsg-attrib-type' prob='0.24'/>
        <state-trans from='emsg-tag-name'
                     to='emsg-attrib-id' prob='0.24'/>
        <state-trans from='emsg-tag-name'
                     to='emsg-close-slash' prob='0.04'/>

        <state-trans from='emsg-attrib-to-end-quote'
                     to='emsg-attrib-from' prob='0.16'/>
        <state-trans from='emsg-attrib-to-end-quote'
                     to='emsg-attrib-type' prob='0.16'/>
230     <state-trans from='emsg-attrib-to-end-quote'
                     to='emsg-attrib-id' prob='0.16'/>
        <state-trans from='emsg-attrib-to-end-quote'
                     to='emsg-attrib-to' prob='0.02'/>
        <state-trans from='emsg-attrib-to-end-quote'
                     to='emsg-close-slash' prob='0.50'/>

        <state-trans from='emsg-attrib-from-end-quote'
                     to='emsg-attrib-to' prob='0.16'/>
        <state-trans from='emsg-attrib-from-end-quote'
240                  to='emsg-attrib-type' prob='0.16'/>
        <state-trans from='emsg-attrib-from-end-quote'
                     to='emsg-attrib-id' prob='0.16'/>
        <state-trans from='emsg-attrib-from-end-quote'
                     to='emsg-attrib-from' prob='0.02'/>
        <state-trans from='emsg-attrib-from-end-quote'
                     to='emsg-close-slash' prob='0.50'/>

        <state-trans from='emsg-attrib-type-end-quote'
                     to='emsg-attrib-to' prob='0.16'/>
250     <state-trans from='emsg-attrib-type-end-quote'
                     to='emsg-attrib-from' prob='0.16'/>
        <state-trans from='emsg-attrib-type-end-quote'
                     to='emsg-attrib-id' prob='0.16'/>
        <state-trans from='emsg-attrib-type-end-quote'
                     to='emsg-attrib-type' prob='0.02'/>
        <state-trans from='emsg-attrib-type-end-quote'
                     to='emsg-close-slash' prob='0.50'/>

        <state-trans from='emsg-attrib-id-end-quote'
260                  to='emsg-attrib-to' prob='0.16'/>
        <state-trans from='emsg-attrib-id-end-quote'
                     to='emsg-attrib-type' prob='0.16'/>
        <state-trans from='emsg-attrib-id-end-quote'
                     to='emsg-attrib-from' prob='0.16'/>
        <state-trans from='emsg-attrib-id-end-quote'
                     to='emsg-attrib-id' prob='0.02'/>
        <state-trans from='emsg-attrib-id-end-quote'
                     to='emsg-close-slash' prob='0.50'/>

270     <state-trans from='emsg-close-slash' to='emsg-close-ab'/>
```

```
<!-- Normal message states -->

    <state name='norm-msg-open-ab'>&open-ab-token;</state>

    <state name='norm-msg-tag-name'>
      <token name='tag-name'>
        <generator func='constant'><arg>message</arg></generator>
      </token>
    </state>


    <state name='norm-msg-attrib-to'>
      <token name='to-valid' prob='0.9'>
        <generator func='constant'>
          <arg> to=&apos;</arg>
        </generator>
      </token>
      <token name='to-no-equals-quote' prob='0.1'>
        <generator func='constant'><arg> to</arg></generator>
      </token>
      <token name='to-no-quote' prob='0.1'>
        <generator func='constant'><arg> to=</arg></generator>
      </token>
      <token name='to-no-equals' prob='0.1'>
        <generator func='constant'>
          <arg> to&apos;</arg>
        </generator>
      </token>
    </state>

    <state name='norm-msg-attrib-to-val'>
      <token name='jid'>
        <generator func='gen_jid'><arg>0.05</arg></generator>
      </token>
      <token name='alice'>
        <generator func='constant'>
          <arg>alice@swordfish.ll.mit.edu/test</arg>
        </generator>
      </token>
      <token name='bob'>
        <generator func='constant'>
          <arg>bob@swordfish.ll.mit.edu/test</arg>
        </generator>
      </token>
    </state>

    <state name='norm-msg-attrib-to-end-quote'>
      &quote-token;
    </state>

    <state name='norm-msg-attrib-from'>
      <token name='from-valid' prob='0.9'>
        <generator func='constant'>
```

```
          <arg> from=&apos;</arg>
        </generator>
      </token>
      <token name='from-no-equals-quote' prob='0.1'>
        <generator func='constant'><arg> from</arg></generator>
      </token>
      <token name='from-no-quote' prob='0.1'>
        <generator func='constant'><arg> from=</arg></generator>
      </token>
      <token name='from-no-equals' prob='0.1'>
        <generator func='constant'>
          <arg> from&apos;</arg>
        </generator>
      </token>
    </state>

    <state name='norm-msg-attrib-from-val'>
      <token name='jid'>
        <generator func='gen_jid'><arg>0.05</arg></generator>
      </token>
      <token name='alice'>
        <generator func='constant'>
          <arg>alice@swordfish.ll.mit.edu/test</arg>
        </generator>
      </token>
      <token name='bob'>
        <generator func='constant'>
          <arg>bob@swordfish.ll.mit.edu/test</arg>
        </generator>
      </token>
    </state>

    <state name='norm-msg-attrib-from-end-quote'>
      &quote-token;
    </state>


    <state name='norm-msg-attrib-type'>
      <token name='type-valid' prob='0.9'>
        <generator func='constant'>
          <arg> type=&apos;</arg>
        </generator>
      </token>
      <token name='type-no-equals-quote' prob='0.1'>
        <generator func='constant'><arg> type</arg></generator>
      </token>
      <token name='type-no-quote' prob='0.1'>
        <generator func='constant'><arg> type=</arg></generator>
      </token>
      <token name='type-no-equals' prob='0.1'>
        <generator func='constant'>
          <arg> type&apos;</arg>
        </generator>
      </token>
```

```
      </state>
380
      <state name='norm-msg-attrib-type-val'>
        <token name='chat'>
          <generator func='constant'><arg>chat</arg></generator>
        </token>
        <token name='error'>
          <generator func='constant'><arg>error</arg></generator>
        </token>
        <token name='groupchat'>
          <generator func='constant'><arg>groupchat</arg></generator>
390     </token>
        <token name='headline'>
          <generator func='constant'><arg>headline</arg></generator>
        </token>
        <token name='normal'>
          <generator func='constant'><arg>normal</arg></generator>
        </token>
      </state>

      <state name='norm-msg-attrib-type-end-quote'>
400     &quote-token;
      </state>

      <state name='norm-msg-attrib-id'>
        <token name='id-valid' prob='0.9'>
          <generator func='constant'>
            <arg> id=&apos;</arg>
          </generator>
        </token>
        <token name='id-no-equals-quote' prob='0.1'>
410       <generator func='constant'><arg> id</arg></generator>
        </token>
        <token name='id-no-quote' prob='0.1'>
          <generator func='constant'><arg> id=</arg></generator>
        </token>
        <token name='id-no-equals' prob='0.1'>
          <generator func='constant'>
            <arg> id&apos;</arg>
          </generator>
        </token>
420   </state>

      <state name='norm-msg-attrib-id-val'>
        <token name='id' charset='numeric'>
          <generator func='gen_varlen_str'>
            <arg>0.05</arg>
          </generator>
        </token>
      </state>

430   <state name='norm-msg-attrib-id-end-quote'>
        &quote-token;
      </state>
```

```
    <state name='norm-msg-close-ab'>&close-ab-token;</state>

    <!-- Normal Message paths -->
    <state-trans from='norm-msg-attrib-to'
                 to='norm-msg-attrib-to-val'/>
    <state-trans from='norm-msg-attrib-to-val'
                 to='norm-msg-attrib-to-end-quote'/>

    <state-trans from='norm-msg-attrib-from'
                 to='norm-msg-attrib-from-val'/>
    <state-trans from='norm-msg-attrib-from-val'
                 to='norm-msg-attrib-from-end-quote'/>

    <state-trans from='norm-msg-attrib-type'
                 to='norm-msg-attrib-type-val'/>
    <state-trans from='norm-msg-attrib-type-val'
                 to='norm-msg-attrib-type-end-quote'/>

    <state-trans from='norm-msg-attrib-id'
                 to='norm-msg-attrib-id-val'/>
    <state-trans from='norm-msg-attrib-id-val'
                 to='norm-msg-attrib-id-end-quote'/>

    <state-trans from='norm-msg-open-ab' to='norm-msg-tag-name'/>
    <state-trans from='norm-msg-tag-name'
                 to='norm-msg-attrib-to' prob='0.24'/>
    <state-trans from='norm-msg-tag-name'
                 to='norm-msg-attrib-from' prob='0.24'/>
    <state-trans from='norm-msg-tag-name'
                 to='norm-msg-attrib-type' prob='0.24'/>
    <state-trans from='norm-msg-tag-name'
                 to='norm-msg-attrib-id' prob='0.24'/>
    <state-trans from='norm-msg-tag-name'
                 to='norm-msg-close-slash' prob='0.04'/>

    <state-trans from='norm-msg-attrib-to-end-quote'
                 to='norm-msg-attrib-from' prob='0.16'/>
    <state-trans from='norm-msg-attrib-to-end-quote'
                 to='norm-msg-attrib-type' prob='0.16'/>
    <state-trans from='norm-msg-attrib-to-end-quote'
                 to='norm-msg-attrib-id' prob='0.16'/>
    <state-trans from='norm-msg-attrib-to-end-quote'
                 to='norm-msg-attrib-to' prob='0.02'/>
    <state-trans from='norm-msg-attrib-to-end-quote'
                 to='norm-msg-close-slash' prob='0.50'/>

    <state-trans from='norm-msg-attrib-from-end-quote'
                 to='norm-msg-attrib-to' prob='0.16'/>
    <state-trans from='norm-msg-attrib-from-end-quote'
                 to='norm-msg-attrib-type' prob='0.16'/>
    <state-trans from='norm-msg-attrib-from-end-quote'
                 to='norm-msg-attrib-id' prob='0.16'/>
    <state-trans from='norm-msg-attrib-from-end-quote'
```

```
                to='norm-msg-attrib-from' prob='0.02'/>
      <state-trans from='norm-msg-attrib-from-end-quote'
                to='norm-msg-close-slash' prob='0.50'/>

      <state-trans from='norm-msg-attrib-type-end-quote'
                to='norm-msg-attrib-to' prob='0.16'/>
      <state-trans from='norm-msg-attrib-type-end-quote'
                to='norm-msg-attrib-from' prob='0.16'/>
      <state-trans from='norm-msg-attrib-type-end-quote'
                to='norm-msg-attrib-id' prob='0.16'/>
      <state-trans from='norm-msg-attrib-type-end-quote'
                to='norm-msg-attrib-type' prob='0.02'/>
      <state-trans from='norm-msg-attrib-type-end-quote'
                to='norm-msg-close-slash' prob='0.50'/>

      <state-trans from='norm-msg-attrib-id-end-quote'
                to='norm-msg-attrib-to' prob='0.16'/>
      <state-trans from='norm-msg-attrib-id-end-quote'
                to='norm-msg-attrib-type' prob='0.16'/>
      <state-trans from='norm-msg-attrib-id-end-quote'
                to='norm-msg-attrib-from' prob='0.16'/>
      <state-trans from='norm-msg-attrib-id-end-quote'
                to='norm-msg-attrib-id' prob='0.02'/>
      <state-trans from='norm-msg-attrib-id-end-quote'
                to='norm-msg-close-ab' prob='0.50'/>

    <!-- Message elements -->

    <!-- Body -->
      <state name='elt-body-start-open-ab'>&open-ab-token;</state>

      <state name='elt-body-start-tag-name'>
        <token name='body'>
          <generator func='constant'><arg>body</arg></generator>
        </token>
      </state>

      <state name='elt-body-start-close-ab'>&close-ab-token;</state>

      <state name='elt-body-content'>
        <token name='short-string' charset='alpha-numeric'>
          <generator func='gen_varlen_str'>
            <arg>0.01</arg>
          </generator>
        </token>
        <token name='long-string' charset='alpha-numeric'>
          <generator func='gen_varlen_str'>
            <arg>0.001</arg>
          </generator>
        </token>
      </state>

      <state name='elt-body-end-open-ab'>&open-ab-token;</state>
```

```
    <state name='elt-body-end-slash'>&slash-token;</state>

    <state name='elt-body-end-tag-name'>
      <token name='body'>
        <generator func='constant'><arg>body</arg></generator>
      </token>
    </state>

    <state name='elt-body-end-close-ab'>&close-ab-token;</state>

  <!-- Body paths -->
    <state-trans from='elt-body-start-open-ab'
                 to='elt-body-start-tag-name'/>
    <state-trans from='elt-body-start-tag-name'
                 to='elt-body-start-close-ab'/>
    <state-trans from='elt-body-start-close-ab'
                 to='elt-body-content'/>
    <state-trans from='elt-body-content'
                 to='elt-body-end-open-ab'/>
    <state-trans from='elt-body-end-open-ab'
                 to='elt-body-end-slash'/>
    <state-trans from='elt-body-end-slash'
                 to='elt-body-end-tag-name'/>
    <state-trans from='elt-body-end-tag-name'
                 to='elt-body-end-close-ab'/>

    <state-trans from='elt-body-end-close-ab'
                 to='elt-subject-start-open-ab' prob='0.28'/>
    <state-trans from='elt-body-end-close-ab'
                 to='elt-thread-start-open-ab' prob='0.28'/>
    <state-trans from='elt-body-end-close-ab'
                 to='elt-body-start-open-ab' prob='0.04'/>
    <state-trans from='elt-body-end-close-ab'
                 to='norm-msg-end-open-ab' prob='0.4'/>

  <!-- Subject -->
    <state name='elt-subject-start-open-ab'>
      &open-ab-token;
    </state>

    <state name='elt-subject-start-tag-name'>
      <token name='subject'>
        <generator func='constant'><arg>subject</arg></generator>
      </token>
    </state>

    <state name='elt-subject-start-close-ab'>
      &close-ab-token;
    </state>

    <state name='elt-subject-content'>
      <token name='short-string' charset='alpha-numeric'>
        <generator func='gen_varlen_str'>
          <arg>0.01</arg>
```

```
        </generator>
      </token>
      <token name='long-string' charset='alpha-numeric'>
        <generator func='gen_varlen_str'>
          <arg>0.001</arg>
        </generator>
      </token>
    </state>

    <state name='elt-subject-end-open-ab'>&open-ab-token;</state>

    <state name='elt-subject-end-slash'>&slash-token;</state>

    <state name='elt-subject-end-tag-name'>
      <token name='subject'>
        <generator func='constant'><arg>subject</arg></generator>
      </token>
    </state>

    <state name='elt-subject-end-close-ab'>
      &close-ab-token;
    </state>

  <!-- Subject paths -->

    <state-trans from='elt-subject-start-open-ab'
                 to='elt-subject-start-tag-name'/>
    <state-trans from='elt-subject-start-tag-name'
                 to='elt-subject-start-close-ab'/>
    <state-trans from='elt-subject-start-close-ab'
                 to='elt-subject-content'/>
    <state-trans from='elt-subject-content'
                 to='elt-subject-end-open-ab'/>
    <state-trans from='elt-subject-end-open-ab'
                 to='elt-subject-end-slash'/>
    <state-trans from='elt-subject-end-slash'
                 to='elt-subject-end-tag-name'/>
    <state-trans from='elt-subject-end-tag-name'
                 to='elt-subject-end-close-ab'/>

    <state-trans from='elt-subject-end-close-ab'
                 to='elt-body-start-open-ab' prob='0.28'/>
    <state-trans from='elt-subject-end-close-ab'
                 to='elt-thread-start-open-ab' prob='0.28'/>
    <state-trans from='elt-subject-end-close-ab'
                 to='elt-subject-start-open-ab' prob='0.04'/>
    <state-trans from='elt-subject-end-close-ab'
                 to='norm-msg-end-open-ab' prob='0.4'/>

  <!-- Thread -->
    <state name='elt-thread-start-open-ab'>&open-ab-token;</state>

    <state name='elt-thread-start-tag-name'>
      <token name='thread'>
```

```
        <generator func='constant'><arg>thread</arg></generator>
      </token>
    </state>

    <state name='elt-thread-start-close-ab'>
      &close-ab-token;
    </state>

    <state name='elt-thread-content'>
      <token name='nmtoken' charset='numeric'>
        <generator func='gen_varlen_str'>
          <arg>0.05</arg>
        </generator>
      </token>
    </state>

    <state name='elt-thread-end-open-ab'>&open-ab-token;</state>

    <state name='elt-thread-end-slash'>&slash-token;</state>

    <state name='elt-thread-end-tag-name'>
      <token name='thread'>
        <generator func='constant'><arg>thread</arg></generator>
      </token>
    </state>

    <state name='elt-thread-end-close-ab'>&close-ab-token;</state>

  <!-- Thread paths -->

    <state-trans from='elt-thread-start-open-ab'
                 to='elt-thread-start-tag-name'/>
    <state-trans from='elt-thread-start-tag-name'
                 to='elt-thread-start-close-ab'/>
    <state-trans from='elt-thread-start-close-ab'
                 to='elt-thread-content'/>
    <state-trans from='elt-thread-content'
                 to='elt-thread-end-open-ab'/>
    <state-trans from='elt-thread-end-open-ab'
                 to='elt-thread-end-slash'/>
    <state-trans from='elt-thread-end-slash'
                 to='elt-thread-end-tag-name'/>
    <state-trans from='elt-thread-end-tag-name'
                 to='elt-thread-end-close-ab'/>

    <state-trans from='elt-thread-end-close-ab'
                 to='elt-body-start-open-ab' prob='0.28'/>
    <state-trans from='elt-thread-end-close-ab'
                 to='elt-subject-start-open-ab' prob='0.28'/>
    <state-trans from='elt-thread-end-close-ab'
                 to='elt-thread-start-open-ab' prob='0.04'/>
    <state-trans from='elt-thread-end-close-ab'
                 to='norm-msg-end-open-ab' prob='0.4'/>
```

```
    <!-- Normal message close tag -->

        <state name='norm-msg-end-open-ab'>&open-ab-token;</state>

        <state name='norm-msg-end-slash'>&slash-token;</state>

        <state name='norm-msg-end-tag-name'>
          <token name='message'>
            <generator func='constant'><arg>message</arg></generator>
          </token>
        </state>

        <state name='norm-msg-end-close-ab'>&close-ab-token;</state>

        <state-trans from='norm-msg-end-open-ab'
                     to='norm-msg-end-slash'/>
        <state-trans from='norm-msg-end-slash'
                     to='norm-msg-end-tag-name'/>
        <state-trans from='norm-msg-end-tag-name'
                     to='norm-msg-end-close-ab'/>

    <!-- Start state distribution -->
        <start-state-dist>
          <start-state name='emsg-open-ab' prob='0.25'/>
          <start-state name='norm-msg-open-ab' prob='0.75'/>
        </start-state-dist>

    </state-machine>

    <!-- Character sets -->

      <charset name='slash'>
        <symbol name='/' prob='0.9'/>
        <symbol name='' prob='0.1'/>
      </charset>

      <charset name='open-ab'>
        <symbol name='<' prob='0.9'/>
        <symbol name='' prob='0.1'/>
      </charset>

      <charset name='close-ab'>
        <symbol name='>' prob='0.9'/>
        <symbol name='' prob='0.1'/>
      </charset>

      <charset name='quote'>
        <symbol name='&quot;' prob='0.9'/>
        <symbol name='' prob='0.1'/>
      </charset>

    </test-config>
```

Appendix B

PCFG Description

This appendix presents the PCFG description used by the random message generator to create test messages for the Jabber Instant Messaging server. As discussed in Chapter 5, the PCFG desription consists of productions that define *non-terminals* in terms of *literals* or other non-terminals. The name of the *non-terminal* appears on the left side of the `'-->'` production symbol and its definition is on the right. All elements in the definition are instantiated, subject to choice and repetition rules and probabilities, then concatenated together to create an instantiation of the non-terminal. The elements in quotes are *literals* defining actual characters that appear in the output.

Several constructs exist for repeating or choosing production elements. The vertical bar (`|`) separates different choices — only one will be instantiated at runtime. The probabilities for each choice are specified on the next line, in the same order as the choices. By default, all choices have equal probabilities. The square brackets (`[]`) designate the enclosed element as optional. The probability for instantiating this element is specified on the next line. The curly braces ({}) signify that the enclosed element should be repeated. The repetition count is defined by a normal distribution, the mean and variance for which are specified on the next line. If a negative value is picked from the resulting distribution, it is treated as 0.

Each production in the PCFG description consists of a single line terminated by a semicolon. The appropriate probabilities or mean and variance values are specified on the line following the production, in the same order as the elements requiring these parameters. To increase readability, these values are aligned to the elements that use them, and some productions are separated by blank lines.

```
start -> message;

message -> empty-msg | normal-msg;
           0.2       0.8

empty-msg -> open-ab 'message' {space attribute} slash close-ab;
                                  2 1

normal-msg -> msg-start-tag  element  {element} msg-end-tag;
                                        2 1

msg-start-tag -> open-ab  'message'  {space attribute}  close-ab;
                                       2 1

attribute -> from | to | id | type | lang;

from -> 'from'  equals  quote  jid  quote;
to -> 'to'  equals  quote  jid  quote;
jid -> string at string slash string |
       'bob@swordfish.ll.mit.edu/test' |
       'alice@swordfish.ll.mit.edu/test';
0.8    0.1    0.1
id -> 'id'  equals  quote  nmtoken  quote;
type -> 'type'  equals  quote  type-val  quote;
type-val -> 'chat' | 'groupchat' | 'headline' | 'normal';
lang -> 'lang'  equals  quote  xml-lang-id  quote;

element -> subject | body | thread;
           0.4     0.4    0.2

subject -> subj-start-tag  string  subj-end-tag;
subj-start-tag -> open-ab  'subject'  [space lang]  close-ab;
                                        0.5
subj-end-tag -> open-ab  slash  'subject'  close-ab;

body -> body-start-tag  string  body-end-tag;
body-start-tag -> open-ab  'body'  [space lang]  close-ab;
                                     0.5
body-end-tag -> open-ab  slash  'body'  close-ab;

thread -> thread-start-tag  nmtoken  thread-end-tag;
thread-start-tag -> open-ab  'thread'  [space lang]  close-ab;
                                         0.5
thread-end-tag -> open-ab  slash  'thread'  close-ab;

msg-end-tag -> open-ab  slash  'message'  close-ab;

xml-lang-id -> char  { char };
               5 5

nmtoken -> number  { number };
           20 2

number -> '0' ... '9';
```

```
string -> short-string | long-string;
                  0.75    0.25

short-string -> { char };
                 100 100

long-string -> { char };
               1024 1024

char -> 'a' ... 'z' | 'A' ... 'Z' | number | '-' | '_';

control-char -> '/' | '<' | '>' | '=' | '\'' | '␣';

space -> '␣' | ''    | control-char;
         0.9    0.05    0.05

slash -> '/' | ''    | control-char;
         0.9    0.05    0.05

open-ab -> '<' | ''    | control-char;
           0.9    0.05    0.05

close-ab -> '>' | ''    | control-char;
            0.9    0.05    0.05

equals -> '=' | ''    | control-char;
          0.9    0.05    0.05

quote -> '\'' | ''    | control-char;
          0.9    0.05    0.05

at -> '@' | ''    | control-char;
      0.9    0.05    0.05
```

Appendix C

CRED Error Messages

This appendix presents the error messages that were produced by CRED for the overflows found by dynamic testing with code instrumentation and random messages described in Chapter 5, as well as those found by the adaptive testing system described in Chapter 6. While the length of the overflow varied between different messages during random message testing, these error messages illustrate the overflows that were identified in `jabberd`. Messages that cause these overflows during dynamic testing with random messages can be found in Appendix D, while messages created by the adaptive testing system can be found in Appendix E.

The following overflow occurs in `jabberd` when the `user` portion of the Jabber ID (JID) is too long:

```
jid.c:103:Bounds error: strcpy with this destination string and size 1045
          would overrun the end of the object's allocated memory.
jid.c:103:  Pointer value: 0xbfffe040
jid.c:103:  Object 'str':
jid.c:103:    Address in memory:    0xbfffe040 .. 0xbfffe43f
jid.c:103:    Size:                 1024 bytes
jid.c:103:    Element size:         1 bytes
jid.c:103:    Number of elements:   1024
jid.c:103:    Created at:           jid.c, line 81
jid.c:103:    Storage class:        stack
```

The following overflow occurs in `jabberd` when the `host` part of the Jabber ID (JID) is too long:

```
jid.c:115:Bounds error: strcpy with this destination string and size 1033
          would overrun the end of the object's allocated memory.
jid.c:115:  Pointer value: 0xbfffe040
jid.c:115:  Object 'str':
jid.c:115:    Address in memory:    0xbfffe040 .. 0xbfffe43f
jid.c:115:    Size:                 1024 bytes
jid.c:115:    Element size:         1 bytes
jid.c:115:    Number of elements:   1024
jid.c:115:    Created at:           jid.c, line 81
jid.c:115:    Storage class:        stack
```

The following overflow occurs in `jabberd` when the `resource` part of Jabber ID (JID) is too long:

```
jid.c:127:Bounds error: strcpy with this destination string and size 1025
          would overrun the end of the object's allocated memory.
jid.c:127:  Pointer value: 0xbfffe040
jid.c:127:  Object 'str':
jid.c:127:    Address in memory:    0xbfffe040 .. 0xbfffe43f
jid.c:127:    Size:                 1024 bytes
jid.c:127:    Element size:         1 bytes
jid.c:127:    Number of elements:   1024
jid.c:127:    Created at:           jid.c, line 81
jid.c:127:    Storage class:        stack
```

The following error occurs when control characters '@' and '/' are out of order in a JID:

```
jid.c:173:Bounds error: in strlen, string argument is a string with size 962
          overrunning the end of object's allocated memory.
jid.c:173:  Pointer value:  0x81c003f
jid.c:173:  Object 'realloc':
jid.c:173:    Address in memory:    0x81c0000 .. 0x81c03ff
jid.c:173:    Size:                 1024 bytes
jid.c:173:    Element size:         1 bytes
jid.c:173:    Number of elements:   1024
jid.c:173:    Created at:           nad.c, line 79
jid.c:173:    Storage class:        heap
```

The following overflow occurs at startup of the `s2s` process and is due to an incorrect indexing into the array (off-by-one):

```
main.c:129:Bounds error: array reference (40) outside bounds of the array
main.c:129:  Pointer value: 0xbffff5e7
main.c:129:  Object 'secret':
main.c:129:    Address in memory:    0xbffff5c0 .. 0xbffff5e7
main.c:129:    Size:                 40 bytes
main.c:129:    Element size:         1 bytes
main.c:129:    Number of elements:   40
main.c:129:    Created at:           main.c, line 65
main.c:129:    Storage class:        stack
```

The following `NULL` pointer dereference was found in feedback testing with a malformed message:

```
jid.c:281:Bounds error: NULL or ILLEGAL string1 argument used in strcmp.
jid.c:281:  Pointer value: NULL
```

Appendix D

PCFG Messages

This appendix presents overflow-revealing Jabber messages that were created by the PCFG random message generator using the grammar shown in Appendix B. The error messages produced by CRED for each overflow are shown in Appendix C. The version of `jabberd` compiled with gcc does not segfault on the overflow occurring at `jid.c, line 115`, but segfaults on the others.

The overflow at `jid.c:103` is revealed by the following message:

```
<message
  to='QPpjQuimP3uFTthazYolOkpEtpYJcSKwOjjXIe-qHXerf0ShxVGqUHvo7fFh
      UkdebSxYlviIiWTkAFYFXFQmgmhRPyILgTi2iTG-nislfmmOSIkNdMaqwy-G
      UUgHBKudVBj__hdzuogoOVQoAhDw_CFBxwMxdkjLrQzDgpfE9sGOcvcoD@NV
      hWwmypwwMrranhtJMtXn-SnAoSzQJvtdodCqfUMSFfpUxhDzCUHlEnDIPDOM
      DDWVXZrhGUhVfhpGJ6rjzwOFIDqzYjDCS2lINpsYSFwdrHskhZm6TDJMjKGH
      PuD_lYQKojYIGjvSBa_QE8cfQC7qfss-sf-zJGnL/skaqhWZefLQgyHKDuAR
      YFvYBdXwIYJrhcMsxZVVlrST-HjjotT_-bgGRIgbdFasxFMiSHoMagTbMeJU
      CoWQDUbuLOTruPBRSteY-UKXdgHsIGEF_GtjBOtJAKVWjPiwzcluOWndKmRj
      lfbqPnmLQCWyhTqDGD_YYyVu-pTzZsfwQDuHMdjRnU8nFzqq-noUokqEAXUv
      USpCMmumvKgUCadlinswdiHCk9BXVuyZUr6GkrhiudJaZh'
  from='DGRsFRUvhoBcIluKSzRsAnXoQfODxaBCtBdzRFmCKCNr_FeKWRMWcSifXy
        ctjlOlzjgjETFUQYIjFieO7PUpDlYWvvf6cBufjmyqyRCVIUciaHAdnWKY
        VFMKreBAqLFnlMYpQoheJByLueucjdXpmgGgzbbu-KtDvyKpGyZALvXyzl
        PdMCL_UMRivaFGEug-mXZGVhzyWVLuIlkvdwD-qlqYj-idELfkVFOCFZXy
        eVVOxNTRMZwsGuxoLMJlcbFAGA7QqioOwgPhGNnuhj_NhXKkZLmmdTo-gf
        m-nAGHrRBkjESX-HdYpTeFpnfuXXRQQJ4tGCIrUCotaVm4HHODepD_ICBc
        RU_Yzer5zfOHvrWbDihfOncIjC_Gepj_TdMOtNeeLWASCxlQadYLKGmdqD
        jEpWDylUymIG-mSMZGLaIMyxZvqoHkycRkIXbSBkQ_yyBsVNIdctORFovz
        ZZuvpVsiqNL3YrWOaiVpaEK-whbvTbnhGg1OMdDVQ-gROxZaqUtgvqVVYU
        gBy-C9fEsQYULPbAAaZPeICCN-wclnI_Cjxwuhd_aiXgMLTfcZLUMNNJP-
        szttDwXvHZBD_iHtkCywEqXexey_WvwDOVNmHpnF2wFwHtMDKPUKlpiWHj
        ckWmkGV-XBRQIJrIyXkgclFmGmzDfrvE_FvsS1jxIKtIvRiHvwKUOkLHqx
        sB1bzicVjwjKZwdtvcFBlcnkfzmOYdNKrnNEUnlRXX7iXHnXHpccakqSDp
        VGMj1n1-jOCdrhLUKSHABP_AueAoLVdJQwjFGMcjwF2gXGpNudPpBCGXUf
        vzoFcU_WzKVtAVFoAsWjIv-FhizFcdudpiZSeluibdmJRApRaRBRwCgsHa
        OwK-zFoVvEx-rQrr7LAafGANdJzmbNXCuicgjqvSzXsq-NWAuTzzOCAIRK
        jNkZzFUgpEWyG_srIuoh_KeZuVoTCjgFJVDhaJTzEvLXZggwjFQI4YmvWz
        MLlBWh-yffOunXtXClaCjhOEsXVzoGvxuDxoxMV_ZRN_KBwDBoaRIHdUBq
```

```
        HwUIma8F_RGSErKMYBXrOjtbiQoTjan_iICgYqhEWQhJlpnXchDoBezpAt
        zdwgPtmHvliIqOzwSmwQDrgQudWVGxNaXyDcvMUDUfCnAKwZsIdkTMizz_
        hTZMiWCEFbKTpBVjPZtnt-sfZbmJXZIvjoBLHRBJrxMHzixXL-CSHZqcNm
        EZpORTjYjIdnpvnnUtrfedeusPLKdUNOUgRyqL0yjx-WIJDVZ-IOce6ex-
        aUCEdV_aREJThlvbpNseEtCsTGBxCykjEKeaeJPYVgXf6Fx-YmRHcvdn7J
        -WnKhmvwnpxnG-BpmZaaWJFmlDopPkrPyUcJ2R6VTNwimAzvRqkaZHeRMp
        o6@xK_jX-Ozx_iTbeKIdpsKXwG-bI_oUKbXx_pstpGW--uyQz-YCNuMhdb
        YezXBfCjgKyYHFggwdDYpMwlPlrsvYNk-BupmtOAx0JfrLxT-OqWugmANO
        EUQtD-uZTRkS-hIvlUxK/rfhjaALcmC-qwvae-YMtWgIHMoCvdhXbMXUdM
        dmAnk-lwNhyjMAKbgmIKJB1Bj9wG-kSCxINYGwgINoxUwGmAHmMGfeCLNI
        CabwVIPGMbtjZBRyXdBCfaXCzdotBhNGmgTjMAxPKgmpWUtnPPracopQrW
        SAnCfaDAZAubEIQ_uHKJSWGONqsqtRKpCOpxQvDDnikeidUWCEDeDSdvcK
        UFrn-L6qUbaGqWNXuB_cCCpWL-UYUCpyiol'
    id='743286057800388170690l'>
    <body lang='ygJZAueOmN'>
      N_WnuPnfEycvTznU–bhhmGvjJdEmLDhQqMqVVlwIB4zhFVDHJGbmoXZA–dmzpZ
      LvvtwdnqoneU–uYYHKolWgUldyvEv_HNhkOJnxKIzSLZjNDudIVpXFRtFRctEf
      vvPQPaiXtIurPZgiJXcCHCb0oowL_wbjfUjVMJxkwJcrkPKo_FkNynzknLBvcp
      LkgpzkeEEu_rESZ_xPpo1UQPBiuvJ4RkLGVNYKTcQJpNPIKWOABYMvfhvtGutV
      rbTxRnxVVFT_YkTnenGrXINsfIsSbGExfbFtVrnQYoSE_WTNi–YHqUDmeziqoH
      bAkmOGiVpGzjtGm6qAxgXctiumYh–gugSSRWRgRzzl–XOlgQFeFQxwaCpZqcdi
      Uvw–_–SY3SBdYZ7QvQn_BetmKNZuibNzoEqdVLRzrzM_HitrfKLRWlVLzTepeV
      RozXfQaMsCaMqiwltBxKsyQaB–QENwJmNAUd_AdIJYDBeDgFBspHAfHunGDO1F
      JuBwzjMCKOTbDJ–fdzkVuEPjuTTgHEQdoku_FehnhxeOEP3eQJf–_GkTxjRrKv
      mSq0dMdf9ZynzzJAWwZeJaNoQry_hcIuCBAw7WX_fuCxVDZnmtP–lsxpJdgBTD
      vFfXPqGpQZgSsfTbEttEmZosCjMuHNcwiUCZfFwkQFcOzsQAlJOJKvPXJnudiZ
      OhuUD6kxraelUhwGAghb_SrqNRnhJKNzNfYRGBLYr_JCniUQZgGwtbepENseie
      EJUiytdkQfRop_zslOlAxDKvTbjZIEhcOJLVrUJDJZ_lLzipnLx6udtjarmyFx
      KbGtRdCkFKNd–ikxmwGgpSIVGzmGLWOgZFED7RXCkBZsSrpIDMP3hKKPBlWMka
      SNOSbdnuMwyCFCjGrcRBYlraDGUMTtQjSQIXYDMOuTrpiIvBkUMHLRVuLpErHC
      qbf9MPhUM–BEqDempmtVOKWpldlYcrQxgjeRRKOxCdZ_rSCUi_feicSjSOjXLT
      TpbqKAhTSR–k5w_LqPurEQwv–esXXbHOxDvwZmp_GpES_wMCGuLeMLDosvTMK
      Pd_MaXbsbhVfceDzoGgDnRwrubAGNV_oRYPWnuwOIgQIZThqMZnQEUubnvkEvX
      –NXvthyj–sEFFctQIqdrXTrbw_zaLgp4rBxRkmjVJWxZwTUBJIU1PkJmexWODC
      EK_f
    </body>
    <thread lang='SzRbmjjxdMrAE'>
      95991868026597240014825O
    </thread>
    <thread>
      98712048124308507991660l
    </thread>
  </message>
```

The overflow at `jid.c:115` is revealed by the following message:

```
  <message
    from='amTrHVDBIPR_DxpIPZKbUupxfAEdLycjmwxLaJT-uMWAdBXdeFvfYuxmQO
          AwOpSvxUaeEdwesyRZjQZ-uTKQpJCtYLUKFHmtxAgFDbabNIE2cJzvDokA
          v5slXAZazLzyUoDkih1xSFW7Dgxojh@CBqoHtqYq-wdhOH-pzWRJfak_xu
          ipeI9cCGSuUFfviULFqijkYkxXunnF_o7vgLNAPKt-soNVIaPXyZpKjQzt
          duK-eCoXcxq4XEHlThVpaue_lJFOEvEIvcmCoJZIUQVQCoakqthbr8uIbg
          SyDrZNBdUAzOz\_KlekAzytQlURptyMhqmmGreBZVC_RiXyxmP4wlqFSzw
          VeWnDPMV_kMrHUlYKYyyXWvXlGVUtEzkfq_BNYYMSWwtQBLIhvzWBGK_-j
          xxzZCM-HOgbKWyxxUXQligAEosuYfXocHagzlejRvNPggujQwQFbCITxTo
```

```
        xOn5MVHyWFV_XKbxlyaSMqFJFWihUPCHsaPd-xjp--ucivHdwerXMOLEjM
        fqAZnDEeYzmzyWfeBonnuVZUvW5lIXYm-QoKnxDgXYRlLCVxemeKecOsVI
        WwGRSVHeUHbYAOszljweVHmTAAOWlTRSKgXhovCEaN9OVQzVrpzNOBmHLc
        OfNFMt9yZWKLogbh-uupEJYEPvCTozt7d_VRzF-ZtszrDxclEL_DZbGXiz
        dMizTopWDYxJqemReZgxXkjRhutIdZArcavNDPIl2FFPfBYsQSTgUqcJgs
        yb9lVlDLoSsDpYbxw-pzRbpdtgNGcTrlF-qSdCdAtKliKyJviDPVCrUuzF
        SgVSugfKBUaXNDZGRRCdCoGcsPcojDia-AHWVpJvZSN_tPTzDJBHEitbRB
        _naTEV-rRJkRhLslC4SMIrCCsSYGlwn5TPMYI-llzNRktumIShpdHzvUEK
        _tOVfXPNghAgUUFulQEknABuG7HXl_YbnrhErVPGUzCPqLUqRHyHbrEaEw
        nf8BFJKSlY_Inj3sQmOcMbwfZoLN-pNpLzeCkwdljLrEQ8zeNCMphT_Pph
        nYSQOX-_vqRITdtn-bH-GvvGiApnkZLBqzVpGKmQKEpzgUhEZCanaJzAup
        UjiXaersFQKSpazsadap9dONreo45Giac_XqxBrpSqzMqhCdoQQas_pxJE
        Tgte4-oRMwRFadDBdfocfEr1B_BZPxEL_MnDFbQQjRRmRgCPXX3qKDworM
        HpNYlnzIFRRmOeeOrQzmWqFEhB_KzclThdVzWWwKtbftArNmrIkhCCvyAz
        ZCxIufKavJaTFDBf-wxho_xcmfj-6CJxs_fTxsV3DFxljhbp-AtuUoQXzS
        s0yAPnb6aWcwXbCiMuzLxe-VXgk0BhyATMdAxWYgSL_cOzRHTesdNtfrl_
        qUElpjMwSkMb-bSPbIzQrVwAKOAM_9uJuBWfJrO2hYhwcOQ7kRi5bCpjQr
        BELbqcfqeYpEwPcuFnMlpBnbrWJGwKIRbcOKfZhyTjgYHyvnjHxJcxuQYq
        VNyEqQrdFCkQpxi-UtPYadfKZEnFK-LncCfOpy5DqjNqeJEPNtiyzccZCm
        TmQk8q-E-ymsX5iuIGyNvgW_CHUeWkOUksvqWknnDXVhCwuQUsPKZRmDmg
        OlktWeiGGJKogoqa_kLncBItd_SUADiaYxjZpYU-yFhTXryQzjWpcnnFN_
        MVWgMLHpz_oBeucPxmidcHkHwdAoRtKXgaPILGtrn_ZvYBPQdLkKuVMQEJ
        eXa2eU0sUZGnwSexX_bENPZ-yN6JmLDHbb-ZVxWXpENrTRUErodNGiNBee
        WYVeDgFPxJtUaBCeRXadF9ShYzwBYEn9tgkdt_VDsKWRVRqn_-pruKzymR
        vEIhfjwxkwkXjTsENCpWODObSEnYQFTf6ApRuvfTcfjYljOOgJa-jPYXqu
        HTzpOqoECCxbUnFJuFEtXSIqWAoDEyezo_xSdDMESEpDHsdQEucDswnjEJ
        aBXZVEJpFnDwCHCY_tdciIzziXzHIWwwESMA7X_XXQXEJdOhQCEpALSll-
        Ei8CUUYUW_C_wsnzhxvJRxrzC-uWaFxKc-gRKieRYfMMnvmFNf_OjO-QiZ
        szNbqYiZvCAFIQtFvf_fwipUe9uDhC'
    type='headline'/>
```

The overflow at `jid.c:127` is revealed by the following message:

```
<message
  from='BpTeuRVPNBdEQwTeobAvpoRQwBjdXVfyKvecwIfnYfKozCsfxZqKDg8uM_
        yLRSxzi_NxFaxSZbJoM-GZNNaxuudm-rdsQEQjCsDSHtPxoSgYt1KQDj_f
        BXfDNVFGhzEytvPgVFzzWSWeSMepkCQbtDhfpgFWsnHZIvEQSdiIOWrtMm
        CH-pmCStPUArsLDOEYgAGKhAbLYAszN3ulwPIBuAnlCdTEDXljIIU_AFLk
        ubE_WaSJXNIyosOM7tbEQuYyLUiixEmVjgpZMZxQI-V-vtxKOtzvLgvmtE
        ARLM2uoqJzDJ_pkPDmqaVCantGiwLLNRGeROs-ai3dknaGntgjXUXWMtts
        CyFdYbLKZVvarHkJgBQsr_WfnDuMVTTCdpAZhynu_SyxgwSupDzOmUuZfx
        UOIMwfoqecBDUbPQLykWfMtICUmsLaQdjSYjyvazWkG-EgBYBSyndBrKbY
        q3sHVLLuKDhsJEmucalhkJlbNbIWtOtQFEdejdlljOFimUTgFfouxUER-_
        YMarsDkNByOg1BWlaVNdDbS-_pcCReAuhyNy_OHToyAjEgNSCVCndIbccV
        2UryFGgkcGwfIAzjxIUwLThJApCtBoLZKpq_DQhKUtlqWoiE_KbrcNMhSG
        qaOGEwIRtNCdEMMBPLlP_aWdESgnOa3WdsniVZFaPnyY-rPjbvaBKCk_do
        ZEsSilWqXIindbTKGHVBwpnqAmmBzABD-ioGPYBkLiFWULNgHbmqbxyXzJ
        GeJHOgoLVynGuLWXJYi_Gej_mqMHtrtpnaUNtkUXAevHVMXnSUbrtdvtCE
        _6PIOiKhDdpIDZkHErKSlwzYtd-fKtgzNosyK5-ezGdSLbtYV-unFjHVGc
        IFPlbxESRjDOVOkDOHFKgObpVgbFVfwuBuUUuBXwkXfCzgHzaH4Fhi2klS
        AYWofvyamPsp_rApIEf5Vj0e4enJ-ieZkLcuUcfxiKwmrP5gkjJf1vVkdL
        vPzEHMrWnhmWCjzMuFICrRfFBwrUyVLCY-VMAxfdagrcaPevlWoEFzcPqr
        JDyLmQcXMoEYbJTzlfvEOBEeJBUpLVpldMPLpEINHJfXHdaCd-REAzBdgg
        tlbzPuN-dEo0-qnqlSptOBkDTjJ2LSnFGVTVXHPwQRSWBsdk_nVN__szSX
        OTfTmUGQLPTvQXIBUPQEZoXOkVKaTLYFAyyilOaonGkBaQQYlElpZsAAmF
```

```
        Lk0dneFmnwcKgqoEbnxhIDQpwDOHgyrL-fSOKDegf0nxnAHrcRkWVpxBmd
        EbbuDUsI_QJEpfewvOzr-PcPpKqqOtsIJYPPvgmC5bQySpnbuvUfCZamKp
        bQbQMXLjRFyfjfJqR_cxgGbQtITWTBpKSfMZJwnjIVEBEdfdgONVKBgdqo
        wC-ebCO-MSTTWcjvguPvQAIYqEUDfWNbvEiRLwxHzRMwZpijjSJJpLpiHE
        qgbVZsiQxlFJVJAxQeInbctbRoGPEZVBfvgK9I_YACKrZlHMPwNiUkMpmo
        ktBKmLrpGEOavEniVO_e@C-orjbOUAhDPSawMTQfJlZIBsgFSddSiZvEXk
        BrNrAjnJeLSifBbBIOPQOqpYLIXjd-TCdx-QI-IOM-oTnndyEBhVW_Drlq
        qdD_wekORhn-DGUJxduNgEpPSFIzuehzYFLTflhsPPjxnHqRCGHAWHnOma
        CNGNxwIYseLmJtoAb/hRNpgJG_FCpdHisJxXitVYFuUnNOUtudatIRiNBw
        nWuLwCaTplkcKoIanPotjMhwpBgWJLulGEHpIkGWjXkksDOUtGqCSEphnW
        PDgLpAC-bNHOPcT'
    to='IBxWU-iiYpCowfLakOBRMFTFYHQENXZPHQnkAPGPeHfbzyFEFpLpGMyUhoGW
        WHJSeGzgoDPSBKcOa6DcAAhmUEMLi_CTpRpTtIgYHsNBVabeqlZBFFZXwBxk
        bnSljQD-fSfZbmdYtdERjLhwzXPuZN-OJTfXntCSqtudoWXtSlijkunWPDRM
        mgLlCjtGRYj_GDL_RC@xLSAgOexqVmJOBRCnrnWIlWmab_oIP-qfQYHpuk_y
        UFDcIsgYlRHaiwcnosUFc1KULSODe_FBbxmNkDgmrIY-IxksZaQuGHdHPCRB
        JlaTbkIMOCOlvPvhXtI_9hAESCUgmDUYYxvnosXZwCCQsEn-ZzlWVwGxxzs7
        KsXHfTCNanX_nROp0MTSbWGVevTYiKVNdhv7a/zbIswEcqlGsuweWjwajiRo
        tuftvZuoVWVBQhyYwvjXBdynEvJcbAdhhxkBVuVFQIlPTrGXpABhzJdMEUGl
        aeWVKBSlgUGmBWpa8lFctZFmClWGZuOtOk_gs_utcSBvbwSoMFhqSyyIAiXf
        RMaIbGCdiscIlNEjxOGDQByScPRwzRsHpiIFFhM2lLwE0dqj--GFJFJNFAaU
        IU8eDNGtGLFYAmFCScMuyIgmxchnaFrN-PvsdTlYQAVD_AhMHMZ_zzAzlRks
        V_qoOAfXKemSYjKIqTRqxTVIZT_sc-noJuKZwtwHZNMsKBqsBV_GEuDLDsYT
        zYYbylJO3WajxxxsgDl-dMTYYOByKBEWVLCCepLPEBJ1KgxzcutTgluYlsvD
        dFuvKZNtpBapZRwPLpIqulcBkSUBYUwNnCfXOYFA6LGbfqRKiotQUwhLhJzS
        CVinCFBXvXZWhdtJrXxBzq_RxLRhGl2Dwod-FLEaEsm-HEVISFCVcmd_bDVX
        CAjeUVnjXxlOYthnzkLuiEqRXHzCBkudDB-YBJvkiPGug_pPjWTAOqYBZr5Q
        ZtErHswnFb-qEcrdXsip8gwswxOjbVAuBPZoYXphXklONTOwocQZUeMVvRqd
        FhKPCqkIjaiZZB-wnxoUsYePhmWtpisjQnMtaIuiz1XSmwsYuPddWW-JPyGK
        MofknXQrIsyQgX_AKfPikmRGFJDZXUrKAlqFKItIibjBLiCEQvMdcNKimnYf
        fAkRRIIdNNWZMzjERt_lPLl9IsyOUuYOfEXHJeRB-tBCChnNWGxnEZBuiNXZ
        iCEQyUQN--nBbRpMKrsMKesYYBNRhGScSFCCinPJALJNeMWINnWoNud_VajM
        dwFCaYBHcAlrnLuHpZHACdRDRRUouojFVhanurlxD-zGMoQq_a-jUFXSzXkN
        wiIgRITs_AtRMKGAdNV-inYxFaHWaDlu-UYKrTFvXMWzmKSdiMBhavHhNWqx
        YFyWBLcDVEpHarfuafSttAxJu_XV3XZCoXcjpLXdlPSOGuwVF-FthQ5GIDuC
        BFgpqf-PaRfZWyUEFb_RabcUSUlKPS6Eu-SnWolRZgJZmAVelalkWobVIgOr
        HGLQLfvTglzHIUZskl--zmAaAUexzf8fFygGVrTnAHPBC-oUz-bjlSbMd4xH
        Kn-dZcKFIl1uiHpjvAmfYxKeeCjZcIuZTjso-DdpQabLddWPZZa-KWjfRvqr
        vRtxfMOcryCWDRtcaTBT_'
    lang='iPCFiFocL'>
    <body>
      xqTMhowO_FreKylDJPMvEKRbKYvmajeSsobEvaC_–FKOnqTwtqW–mOL_lpuTOA
      mnUKCQZsbGVJAbzwTYOtcjSS_CjczEGpyBa–WepFMuRdRNGn
    </body>
    <body>
      YtoYwGTXnZmTsdzWOaHSkiFWJMC–JW–wRSBgJFVcZfeSsOIKIGRNXkODYrRIgY
      FlVfqzQ_Q_GNYupPWy–bBBalevjXtBpCXxXhsleqYFEauWLIGpS–JqtJXQnYG3
      HX6TGpnRzPh5SriLsphbZ–rRk–aKFOoOrBMWgjyoFiNRD–MoVvZibNhwXfWfUz
      GJyfzyGtAyW–xCIjbOslSEUUMCnESJLDXpOIbOVuUP–usYsvvlCsFrVTLsj–sL
      XFzsWrVdJNVlgJIS3SUVzYhKrAtMffvHCMYHYZqrLxRVQrAeASbtMwti_UEQGp
      epyNxnJMdDSxInvLCvOAxTqf9qXaWxYqDhZApgUYGYGJTbo_AQiebDObuvJsi–
      PkcFzIMFjE6Fg6GVnKBlvadiITb_QHkmazEefXlMpdhESgfHKpTBJhGorNICht
      TDHMTFuFLwHuaUJsBiKYwjRMwikdOGbfUSPrhBkbxbXlGSoInIUF9RRuJJUlNS
      oi6FHsRLewnGoPNzp–lUDlUmvHB6iDyQ_aLnUzXUYvMWsctNAjwjlWJNAUKubk
```

```
    WhvMaMrvJLjzbDWI–TXBjW8kMDKSAySpnhYntJG–qozZinkbdT−6hYwfnyzpOx
    DHUnbNpu–lMpGznawGQrImdMyKKvZBtSwr2EgHSwRrdewrthJr−etnjKZwIrEp
    _o_KURhrxq_rkCpYdGUpZyuCNBqZt–GweAiKdpmpmwTDZM4ZPGJLhAiwSvFJJb
80  UbFJhcu_MZcIiXQcF8idcHQ–TTPGyF_pMzqazAkk3YLaFvpDvBDNXKChPtsfyJ
    –HKfUSCFBgnDRtMLtMYlFIvlqzWaPtaEEyEhphXnVWvVSCap_vASmBITBbsLgS
    SablqcySmPoKBBIiYBSnRpTw_GqCSCaKHushtpnGFUOWBCALllVCGhdnuFw8Tg
    EwZcIjw3nzcQkgVkzvc0qOHfDpBccY_hFZeRavmmodZ–icTjOfaOrLKbHoYONN
    eT–jmjonxaQHBHPftQuFdT7sjKUIhrOJpqZDDDzPpaYcbODrlkhWNGPseFCGLK
85  gglSeafibwrTGUo8dSCmBMCjTAHWxhySuh_ytHkewjEDu–c–z_QLCAXtcyvBia
    XzrksavcWfUVNmXASIvbAgtdpabvFl_4mHwdSCtBuMNxdieJWHeruLpIanEogJ
    KlMC0xXILchSladVKHUskLCDaStDnKeDRRFi–cDWfcwsbva_WcjhfbKpASZAhM
    uCepHCdw–jxaRXjxRDsAjw_wwjVpdEaZkoWvNVafDbsJcYQCWukriVHeSncD_G
    UPDYKkIkFgVYfFFTcAxNKDP8pPCStR_JiULxizLiRRvfIcXmyyattGGWisRVvD
90  bFmq–wKuTE–NALFXEDdQPbYtM_WLLyXldTyQkBRHjGeQVfkMSTjXicMVTlzu_c
    vK–oVaGTncHiYnZjCmeKm_nC–UqKjmZUZd–TGWlMqRwdoTVekHgceSGHrqtzXe
    PIqMZMOM5MqiQg
  </body>
  <thread lang='PYGCWqti'>
95  382791306883897559 4929
  </thread>
  <subject>
    wRpjRupeayXTIsidwdyXbKqEyucWewxYZzRcTqhbmpRVbn_x3xT_txeLtoTReU
    zkioRVAAOUOGSd–RV_lAKwlsiPM6kmasFfHldrTzIFz_dmFTnqn–MayAMxUzqU
100 iKGBaZHITHlJrodwvGaypqVKKWfIGLXcbemRoMylVsOjzCV
  </subject>
</message>
```

Appendix E

Feedback Messages

This appendix presents overflow-causing Jabber messages that were created by the adaptive testing system. The error messages produced by CRED for each overflow are shown in Appendix C. As mentioned before, a version of `jabberd` compiled with gcc does not give any indication of the overflows when given these messages.

The original "captured traffic" message used in the adaptive testing system was:

```
<message
  to='alice@swordfish.ll.mit.edu/resource'
  from='bob@swordfish.ll.mit.edu/test'
  id='12345'>
    <thread>
      thread1 thread2 thread3
    </thread>
    <subject>
      subject1 subject2 subject3
    </subject>
    <body>
      body1 body2 body3
    </body>
</message>
```

The overflow at `jid.c:103` is revealed by the following message:

```
<message
  to='aliceeelieecieelaaeellileaceiciacaicaeceeaeaeleliiaacalilcil
      acelclicillceeleccecaaiicllaelceaaeeicacllclacciaeeeccaeccl
      icaealaciaaciclaeciiaclcilaileecleiellleieaececalalecicaeeac
      icaelelaeacceaeclecelacaecelaeaccleiilaceaaaiiieiecaiielilee
      iececcceiilellcailiiiccileaaliaieaieciiaiealleeeiieeaiclaacl
      ecaeieececieilcailciclacalcllicielecaalcccccaiieilleelillcla
      leelccaiacaielilaaeilecalicceaaeaiilaiaaeclcelcaaeleeliaecli
      ceiaeclcceaaciieaalceeceaaiclllcieaialaeeilaiiiiieeceeceliic
      eliclllacaceaieiicailiiaaleilccaieeaaiiiciaaicliillciallaalel
      ellaeceeeicceaeacilelicaiicciiiaaaelellilclecieelellcceaaeac
      illclaaiiclelaeacieelalaaeclccieaiillceliileleiiiiilaaaiacli
      eleeliielceaaeaecillaicicccceiaaiiaeaelelaccaaaieccceliccciia
      iiecaeelciccaiiciccelelilleaaieeaeiiciiicelciaicelilleilcaee
```

```
      caceeaeeccaeecelcelecaceaacacccaaceaiileaeliaeliclaiaalalali
      aleaellaelcliaieleeiceiiiiaclcalilaialiclclaielcicaeeaeaiecc
      lcellacciciicaaciaeaaaeiaeleciiccleicaallcliaaaliiiliclecciiee
      iililleilecllceaclaleaaaiaillicceieecliiceileiaeeeaalciaelcl
      ileceaiiieclecciilcaleiiaiaiecileicialacailllcieiaiclieielac
      iiicececaaeeliielaaellieclcllelaacacaceleeclelcceeaaiellilei
      iaaailclaelcclaacaaaaeaialiccleaililcelcaiellciciccaclaaieea
      iaaliiaccaiaicaiclaclaeeialliaealillaelilicacicilcellelilia c
      eiealieeacaacleeeiliiicacaccalileeieaealiacaaceeaclaieleaela
      lelecciaccaieillllcaeillceeleicliilcaiiaeiceeleceaieacaeeccc
      aaiailcleicceelclelailliaeleciiaaalieeaaaaceaeiaeealaeieeaai
      icelcellieaieiailelelllccieaelciiclllcelcclaiaeieeclleclee l
      eiceiliaileacaecealacccacialleiiliiieilllaaaaccallalaeaieell
      aeeiccacalacecellcieicellcaiaceieaeeiaiaeccceciiaccicaacileii
      iaallaiaielecealcciieaaeiealeacceaaiaceiieaacicaacailiiaacei
      clllaecalaliaiailclaaececlciiaiilicailillliilclaicaclilalcce
      iiaieeecacliaiaeccciaceillaceclililaieeielaeililleececalacei
      lcciielilceeiaailiieecaiiecieaiiaaielacciaiicceacieliiaaeaee
      acececiilecelclllelieclalaciaailaaacicaieelaciciaaaiiicaaeli
      eeeaalieeaeeceieeilcaaaceileaceecealceaeilecillciacielieeael
      eleilaiciliceillacicicccl@swordfish.ll.mit.edu/resource'
  from='bob@swordfish.ll.mit.edu/parse_test'
  id='123435'>
  <thread>
    thread1 thread2 thread3
  </thread>
  <subject>
    subject1 subject2 subject3
  </subject>
  <body>
    body1 body2 body3
  </body>
</message>
```

The overflow at `jid.c:115` is revealed by the following message:

```
<message
  to='alice@swordfish.ll.mit.edu.rds..oi..iiii.s...si....r.do.oi.s
      .....iihi......d.sdf..iw.iro.i...i.ddsd.f..iw..f...hr.ii.i..
      hs..f....dif.......h..ds..hw.dfwfis.ddf.si.hi...h...w.h.hfri
      f..dwifi...iddd.ofhddd.ioddhis...r...fid..did..s.f..d.s.isd.
      ..owi.s.w..wdw.w..i......ii.i...rrird.i..hi.srfd..d.iis.sdd.
      .hd..i...d..is.fh..f.r..rd....w.sdriirsdd..s.fd.r.o..s...dfw
      rs.d..ird..i..hir..d...rdr....s..r.od.oi..odd.i.iswfs..irhr.
      hdid.o.w.idisdd.i.ios...isifd.i...hs...r.rrdwddi.dhwi..di.w.
      ....w.fos..sw...dwrfw.fsi.s.s.si..i...ow....h.os.fs..rr...di
      .d.f.wrd.d.wdd..s.d...ro..srd..d...i..r..wowhii.i.ii..d.hh.f
      .w.o.....owi......rf.....o.swiw..w...dd.ifod.r..i.iifwwfiwdr
      w....s..iods...d.dd.o.w..f.dsr.d.ih..o..diih...orr.s..rdhhwi
      .srwihs.fii.sw...h.sd.is.s.dr...fwdo....hddf.orr..dffw.i..s.
      i.iwrd.r.h.i...o.d..sr...r..s....dfhdsfdowiswh..sssr.do.rd.d
      f.so.w.h..s.ridw....hwhdfrod.wid.ds.fh.wif..dsf.i.sw..r.....
      r...s.dh.r....s..f...i.fddi..d..ido...idsif.....h.issids.fhr
      .r.fdd.io.fs.sfi.....i.s..iihhsw..d.i...iii.dowod.fsisi.d.f.
      r.iiod.ofowi.s.hih.../resource'
```

```
    from='bob@swordfish.ll.mit.edu/parse_test'
    id='123435'>
    <thread>
      thread1 thread2 thread3
    </thread>
    <subject>
      subject1 subject2 subject3
    </subject>
    <body>
      body1 body2 body3
    </body>
  </message>
```

The overflow at `jid.c:127` is revealed by the following message:

```
  <message
    to='alice@swordfish.ll.mit.edu/resourceeuseorerorursueurueeerrse
        seoeeeoeerreurrsrscurueeeuueuurceoscsreuceeecrreuorroreouecc
        ssrccrceueeeesrrccseeueerruuecrrercreeereeuuecreeurrcuoocece
        srreuuceuscsuseursroeeroccerececrosceserreseoroesrooeuescrcu
        eroeuerrrureeuceesuerursorrcrrrurroercesreeooeeooscrrscseuue
        rccoseoeouuceorooerrerrcuseosrrcosrrseurreerrosueocecoerceur
        oresucrrrrsorurescoeseuecrcceurusuoeeecreoreocesuccrrsurrrrr
        uueesereceeeesurcurrusrorcccrurserrcrcrersruousrerecroosoece
        ueoocusrueroccrecerrscureeeccurrereeresesrrecsceeorusrceceee
        rercrocrceressroueserroeuoeeescreusrorrucueecuserorsusrrreee
        eorerrcerseeeeescrrrcuccececocereeeurrrruuecrocsecorcsrerrso
        reeruerrrrrssereooocseucooeoeeururrsrssssesccrsesrreucoereru
        rrrorueroroureurrorecereroeroeesrceroescrrrerroereeesrsuesse
        eerrerrrursoereerrecerrcesorsosrorueeurueeesereercseuoseru
        ercccorcueeruroosoeessueocsouseoccecorccesorrcsesrereccuusee
        ereueeeruroecocuueccuecsrueesuecrsoceorocueosreroeoecuccuroo
        ressoorceeseusccsrsosruereersreruucesssorrerueorucsrcosurrrs
        erruereorsesroeeeeerceerusorcosrrrreoccsureosseeecsrouruersu
        escrrssrucrrrueosouoeresesesseseuuceuercsrrussececruoseucrrc
        rrsrreccerrsoseoeeusesosorrcorrorrcrcesuresurrsoeoeeorsssuue
        eocuserrerccereeeecroureceesersueuooursoeeeuuscorerooeuueeus
        rroruruuerrrserecuoeresersrroreoerureecoeurcrrscresrouorseur
        eeuucooroeureusrusooreoeorsecsoeserrecueuouerrruerorrueuoeru
        eeeeeesoecrceerossrsrsuceeeerurrurssuorreeuoeeueescuucerrcer
        eerececeerserosuoresoreoesuecsersrererrcrcuocouorsserrrcrrrr
        uocruceroceoeeooreererscreoeocroereorucerurrescrsereeeserosu
        sreeerrr'
    from='bob@swordfish.ll.mit.edu/parse_test'
    id='123435'>
    <thread>
      thread1 thread2 thread3
    </thread>
    <subject>
      subject1 subject2 subject3
    </subject>
    <body>
      body1 body2 body3
    </body>
  </message>
```

The overflow at `jid.c:173` is revealed by the following message:

```
<message
  to='/alicelaeialaeaciccalaleaeliiilllllilcalacaeelcceiilecalaceclllcaalaiaclicliilaceeciicilccieeeeaieicccalccacaieaiclecilclccccceeilaeceaciiailieaiieacaaialclcceeliaeclcaceiicciiacacaielceilalliilicaaiaaciellaaililcieceileleailcleicicaiailiiceeeeiiccleccaicecieccaaeeiiiacilailecaaiacielicleilaaaiaeccicillaialeaiccileclceieceiiclcelcaeliiliecaeilicaeiaaieelieliiclclaaiceeailceelaleeliaailieeeeicaliieiacecieailiilialcalallllcaeacieliclelaeacaalealaeilicieiieileclacceaciliclealcaciaelallleceeialelclccleacialllclieililaailillleilllaaeaeeiaealalaeacaalaeicilacieiialeliecaieacacecalilcelcicieiiaiieiaeliclacacaealeaeeeiiiiacialcaaealcilleaialilaecaleclliieacecalalcaieiaelaialcllileleecelieleaieaicailaclcaeeiacaiallliicicicileieaielelacaelcaaliacillliiiaelellcceccaeaiiaeliceaaaciaciclceiicailaceeccaiiealaccallaacieceiiccclcaleaclcieealciilieelcclcleeclclieicelciliaieiclceaiiaeeiieceiaeiilleclaaieicliaceiileaccilclaaaaeelliiacciciacaiilaallaeieiaeeccecеaiaieaceillcecallclcaaacilcacacecilcieaa@swordfish.ll.mit.edureso
  urce'
  from='bob@swordfish.ll.mit.edu/parse_test'
  id='123435'>
  <thread>
    thread1 thread2 thread3
  </thread>
  <subject>
    subject1 subject2 subject3
  </subject>
  <body>
    body1 body2 body3
  </body>
</message>
```

The `NULL` pointer dereference at `jid.c:281` is revealed by the following message:

```
<message
  to='alice@swordfish.ll.mit.edu/resource'
  from='/bob@swordfish.ll.mit.eduw...d.i..wsffih..i.ro..i.ii.r..dh
        .ofi......w.s.isiid..io.d.o.o.w.w.hi..wi.fhrd.ifs..fio.sis
        is.d...si..dfodsi.dwod.o...f...hf.i.dr.rod...iidw....ih..f
        .wi.d.s..ds..fi.hrr.ws..s.df.i...wsd.d.ir.r.ddf..w...rs..f
        dhsi.fd.ssdr.ds.s....d..f.i..d...sdoo.s.hsddihs...d.iiis..
        .d...rdrio..d.fio..doo..shr.s...dr.d......hf..oid..s.d.i..
        .hoo.dwdr.di.offh...dsii.d...s.if.s..si...d..oiof..ww..isd
        ..df.o.d..dw......i...s.o..rf..rh.foi.s.swiwsi.h...sw..d.i
        .d..fid.hdssfhi....iid.d....s.iiw.f..h.ds.....i.s...r.o.so
        hs.o..is...dd..wdssdwfsrrr.rssi.fd..hsss..s..hiih.s.id...r
        .i...osi...wd..ow.iw.ssssdifr.o.sdi.rordfdd..di.dh.oi.s.d.
        s.s..r..w..d.hs.dwhihh.isd.i..fdsfhh..owddhh.f..h.o.s.so.s
        .id..wd.odhi...w.h.wdis...d....iddh.i..oifdhod.if.iwd.s..i
        wsdfisif.h...f.ord.h...d.s.sr..i.rr.hhh..i.i.io....wddo...
        .d.ds.s.rdi..s......sho..dwwwhii.s..r.h.h...d..w.d..ohf...
        ..ds.dh.dsrohi.sow.wd....if.....swf.drii..s..fw.difh......
        ..rssrr.h...si.sdd..sw.iw...dsiwssh..i..s..id.hss.ids..hi'
  id='123435'>
```

```
    <thread>
      thread1 thread2 thread3
    </thread>
    <subject>
      subject1 subject2 subject3
    </subject>
    <body>
      body1 body2 body3
    </body>
</message>
```

Appendix F

Literals Found in `jabberd`

This appendix lists the character literals found in `jabberd` using the CIL module described in Section 6.5. These character literals were given to the tester and used to parse messages into tokens in the adaptive testing system. The table below shows these literals sorted by the number of occurrences in the source code.

Literal	Number of Occurences	Literal	Number of Occurences
0x00	54	0x1A	1
=	5	0xF0	1
|	4	0x03	1
space	4	0xF4	1
/	3	z	1
\n	3	$	1
<	3	0xE0	1
>	3	0x1C	1
)	3	0x7F	1
’	3	0x06	1
,	2	.	1
?	2	0x16	1
&	2	[	1
(	2	;	1
:	2	!	1
@	2	{	1
0x01	2	]	1
0x80	2	%	1
a	1	0x02	1
\\	1	0xC2	1
0x0D	1	\t	1
0x0C	1	}	1
0xED	1	0xEF	1

Bibliography

[1] AlephOne. Smashing the stack for fun and profit. *Phrack Magazine*, 7(47), 1998.

[2] Arash Baratloo, Timothy Tsai, and Navjot Singh. Libsafe: Protecting critical elements of stacks. Technical report, Bell Labs, Lucent Technologies, 600 Mountain Ave, Murray Hill, NJ 07974 USA, December 1999.

[3] Arash Baratloo, Timothy Tsai, and Navjot Singh. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[4] Fabrice Bellard. TCC: Tiny C compiler. http://www.tinycc.org, October 2003.

[5] Nikita Borisov. AT&T failure of January 15, 1990. http://www.cs.berkeley.edu/∼nikitab/courses/cs294-8/hw1.html, April 2005.

[6] CERT. Advisory CA-2001-19 CodeRed worm exploiting buffer overflow in IIS indexing service dll. http://www.cert.org/advisories/CA-2001-19.html, January 2002.

[7] CERT. Advisory CA-2003-16 buffer overflow in Microsoft RPC. http://www.cert.org/advisories/CA-2003-16.html, August 2003.

[8] Monica Chew and Dawn Song. Mitigating buffer overflows by operating system randomization. Technical report, Carnegie Mellon University, December 2002.

[9] James Clark. The expat XML parser. http://expat.sourceforge.net, April 2005.

[10] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 232–244. ACM Press, 2003.

[11] ICF Consulting. The economic cost of the blackout: An issue paper on the northeastern blackout, August 14, 2003. http://www.icfconsulting.com/Markets/Energy/doc_files/blackout-economic-costs.pdf, August 2003.

[12] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, January 1998.

[13] Crispin Cowan. Software security for open-source systems. *IEEE Security & Privacy*, 1(1):38–45, 2003.

[14] Nelly Delgado, Ann Quiroz Gates, and Steve Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, December 2004.

[15] Berke Durak. Geyik: A random-sentence generator, taking BNF as input. http://abaababa.ouvaton.org/caml/, October 2004.

[16] Frank Ch. Eigler. Mudflap: Pointer use checking for C/C++. In *Proceedings of the 2003 GCC Summit*, 2003.

[17] Hiroaki Etoh. GCC extension for protecting applications from stack smashing attacks. http://www.trl.ibm.com/projects/security/ssp/, December 2003.

[18] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[19] U.S.-Canada Power System Outage Task Force. Final report on the August 14, 2003 blackout in the United States and Canada: Causes and recommendations. Technical report, United States Department of Energy, 1000 Independence Avenue, SW, Washington, DC 20585, April 2004.

[20] Jabber Software Foundation. Jabber: Open instant messaging and a whole lot more. http://www.jabber.org/, April 2005.

[21] Anup Ghosh, Tom O'Connor, and Gary McGraw. An automated approach for identifying potential vulnerabilities in software. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 104–114, May 1998.

[22] G.J. Holzmann. Static source code checking for user-defined properties. In *Proceedings of IDPT 2002*, Pasadena, CA, USA, June 2002.

[23] IETF. The internet engineering task force. http://www.ietf.org/.

[24] Yahoo! Inc. Yahoo! messenger. http://messenger.yahoo.com/.

[25] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275 – 288, June 2002.

[26] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Automated and Algorithmic Debugging*, pages 13–25, 1997.

[27] Nick Nethercote Julian Seward and Jeremy Fitzhardinge. Valgrind: A GPL'd system for debugging and profiling x86-linux programs. http://valgrind.kde.org, 2004.

[28] Rauli Kaksonen. A functional method for assessing protocol implementation security. Publication 448, VTT Electronics, Telecommunication Systems, Kaitoväylä 1, PO Box 1100, FIN-90571, Oulu, Finland, October 2001.

[29] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[30] Robert Lemos. Security a work in progress for microsoft. *CNET News.com*, January 2004.

[31] Steve Lipner and Michael Howard. The trustworthy computing security development lifecycle. In *2004 Annual Computer Security Applications Conference*, December 2004.

[32] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[33] Jim Morrison. Blaster revisited. *ACM Queue: Architecting Tommorow's Computing*, 2(4), 2004.

[34] George Necula, Scott McPeak, Shree Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Lecture Notes in Computer Science*, volume 2304, page 213, January 2002.

[35] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[36] NIST. Software errors cost US economy $59.5 billion annually. http://www.nist.gov/public_affairs/releases/n02-10.htm, June 2002.

[37] NIST. ICAT vulnerability database. http://icat.nist.gov/, February 2005.

[38] NIST. ICAT vulnerability statistics. http://icat.nist.gov/icat.cfm?function=statistics, February 2005.

[39] Rob Norris. Jabberd project. http://jabberd.jabberstudio.org/2/, March 2005.

[40] Jeff Offutt and Wuzhi Xu. Generating test cases for web services using data perturbation. *ACM SIGSOFT Software Engineering Notes*, 29(5), 2004.

[41] America Online. AOL instant messenger. http://www.aim.com/.

[42] CBC News Online. CBC news indepth: Power outage. http://www.cbc.ca/news/background/poweroutage/numbers.html, August 2003.

[43] Ed Oswald. Gmail bug exposes e-mails to hackers. http://www.betanews.com/article/Gmail_Bug_Exposes_Emails_To_Hackers/1105561408, January 2005.

[44] Parasoft. Insure++: Automatic runtime error detection. http://www.parasoft.com, 2004.

[45] Ed Parry. Microsoft official: XP SP1 first product of trustworthy computing. *SearchWin2000.com*, September 2002.

[46] P.J. Plauger and Jim Brodie. *Standard C*. PTR Prentice Hall, Englewood Cliffs, NJ, 1996.

[47] Lawrence Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In *IEEE Proceedings*, 1989.

[48] Eric Rescorla. Is finding security holes a good idea? *IEEE Security & Privacy*, 3(1):14–19, 2005.

[49] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel Roy, and William Beebee. Enhancing availability and security through failure-oblivious computing. Technical Report 935, Massachusetts Institute of Technology, 2004.

[50] Olatunji Ruwase and Monica Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS 2004)*, February 2003.

[51] Donn Seeley. A tour of the worm. In *Proceedings of 1989 Winter Usenix Conference*, pages 287 – 304, January 1989.

[52] PolySpace Technologies. PolySpace C verifier. http://www.polyspace.com/c.htm, September 2001.

[53] USA Today. The cost of CodeRed. http://www.usatoday.com/tech/news/2001-08-01-code-red-costs.htm, August 2001.

[54] Timothy Tsai and Navjot Singh. Libsafe project. http://www.research.avayalabs.com/project/libsafe/.

[55] Vendicator. Stackshield. http://www.angelfire.com/sk/stackshield/index.html.

[56] 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.

[57] Watchfire. Ensuring security in the web application development life cycle. www.watchfire.com/resources/ensuring-security-wdlc.pdf, 2004.

[58] Wikipedia. Heisenbug. http://en.wikipedia.org/wiki/Heisenbug, May 2005.

[59] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, pages 149–162, February 2003.

[60] Yichen Xie, Andy Chou, and Dawson Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336. ACM Press, 2003.

[61] Michael Zhivich, Tim Leek, and Richard Lippmann. Dynamic buffer overflow detection. *Submitted to the 13th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, September 2005.

[62] Misha Zitser. Securing software: An evaluation of static source code analyzers. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, August 2003.

[63] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Softw. Eng. Notes*, 29(6):97–106, 2004.