Enhancing Availability and Security Through Boundless Memory Blocks

by

Cristian Cadar

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
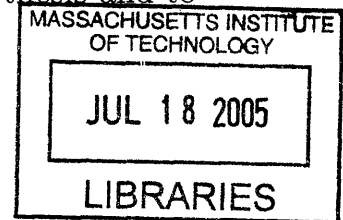
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2004

© Cristian Cadar, MMIV. All rights reserved.

The author hereby grants to MIT permission to reproduce and
distribute publicly paper and electronic copies of this thesis and to
grant others the right to do so.

Author ....
Department of Electrical Engineering and Computer Science
July 14, 2004

Certified by...
Martin C. Rinard
Associate Professor
Thesis Supervisor

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

Enhancing Availability and Security Through Boundless Memory Blocks

by

Cristian Cadar

Submitted to the Department of Electrical Engineering and Computer Science
on July 14, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We present a new technique, boundless memory blocks, that automatically eliminates buffer overflow errors, enabling programs to continue to execute through memory errors without memory corruption.

Buffer overflow vulnerabilities are caused by programming errors that allow an attacker to cause the program to write beyond the bounds of an allocated memory block to corrupt other data structures. The standard way to exploit a buffer overflow vulnerability involves a request that is too large for the buffer intended to hold it. The buffer overflow error causes the program to write part of the request beyond the bounds of the buffer, corrupting the address space of the program and causing the program to execute injected code contained in the request.

Our boundless memory blocks compiler inserts checks that dynamically detect all out of bounds accesses. When it detects an out of bounds write, it stores the value away in a hash. Our compiler can then return the stored value as the result of an out of bounds read to that address. In the case of uninitialized addresses, our compiler simply returns a predefined value.

We have acquired several widely used open source applications (Apache, Sendmail, Pine, Mutt, and Midnight Commander). With standard compilers, all of these applications are vulnerable to buffer overflow attacks as documented at security tracking web sites. Instead, our compiler enables the applications to execute successfully through buffer overflow attacks to continue to correctly service user requests without security vulnerabilities. We have also found that only one application contains uninitialized reads, which means that in most cases, the net effect of our compiler is to (conceptually) give each allocated memory block unbounded size and to eliminate out of bounds accesses as a programming error.

Thesis Supervisor: Martin C. Rinard
Title: Associate Professor

3

Acknowledgments

First and foremost, I would like to thank my research advisor, Professor Martin Rinard, for all his help and advice during my time at MIT, and especially during the last twelve months. His enthusiasm for research, and his courage to pursue new and unconventional ideas have contributed to a great extent to my decision to pursue doctoral studies in computer science.

Second, I would like to thank Brian Demsky, Daniel Roy, and Daniel Dumitran, my collaborators during my Master's year at MIT on the Boundless Memory Blocks project, the Failure Oblivious project, and the Archie project.

I also want to thank all the other professors and students who have contributed in some way to the success of my studies at MIT, among whom Prof. Daniel Jackson, Dr. Ed Chi, all the members of Martin's research group, and of course, my friends.

Last, but not least, I would like to express my deepest gratitude to my mother and father, my sister, and my late grandmother for all the support they provided me during my studies at MIT.

Contents

List of Figures

List of Tables

11

Chapter 1

Introduction

Standard programming languages (C, Java, C++) allow programmers to create and then access memory blocks (such as buffers, objects, structs, or arrays) of a fixed size. An attempt by the program to use a reference to a block to access memory outside the block is considered to be a programming error. The meaning of a program containing such an error varies from language to language. Java implementations, for example, check all accesses and throw an exception if the program attempts to access out of bounds data. The ANSI C standard, on the other hand, specifies that the meaning of a program is undefined if it uses pointer arithmetic or other means to access data outside of the block boundaries. In practice, most C implementations do not check for out of bounds accesses, leaving C programs vulnerable to data structure corruption errors that occur when an out of bounds access to one block corrupts data stored in another block. Because the effect of these kinds of errors is so dependent on aspects of the implementation (such as the layout of the data structures in memory) that are outside of the basic programming model of the language, they can be extremely difficult to reproduce and eliminate. And because they can corrupt language implementation structures such as return addresses and function pointers, they often leave the program vulnerable to buffer overflow attacks, which attackers can exploit to inject and execute arbitrary code over the network.

In this thesis we present a different approach, *boundless memory blocks*, to out of bounds accesses. We generate code that checks all accesses, but instead of allowing

out of bounds accesses to corrupt other data structures or responding to out of bounds accesses by throwing an exception, the generated code takes actions that allow the program to continue to execute without interruption. Specifically, it stores the values of out of bounds writes in a hash table indexed under the written address (expressed as an offset relative to an identifier for the written block). It can then return the stored value as the result of out of bounds reads to that address. It simply returns a value from a predefined sequence for out of bounds reads that access uninitialized addresses.

Conceptually, our technique gives each memory block unbounded size. The initial memory block size can therefore be seen not as a hard boundary that the programmer must get right for the program to execute correctly, but rather as a flexible hint to the implementation of the amount of memory that the programmer may expect the program to use in common cases.

We have developed a C compiler that implements boundless memory blocks and used this compiler to generate code for a collection of widely used server programs drawn from the open-source Linux community. As documented at security tracking web sites such as http://www.securityfocus.com and http://www.securiteam. com, all of these programs have security vulnerabilities related to out of bounds accesses such as buffer overflow errors. Our results show that the use of boundless memory blocks makes these programs invulnerable to these security vulnerabilities and that the overhead associated with using boundless memory blocks is acceptable in practice.

Note that boundless memory blocks have the potential to introduce a new denial of service security vulnerability: the possibility that an attacker may be able to produce an input that will cause the program to generate a very large number of out of bounds writes and therefore consume all of the available memory. We address this problem by treating the hash table that stores out of bounds writes as a fixed-size least recently used (LRU) cache. This bounds the amount of memory that an attacker can cause out of bounds writes to consume.

This thesis makes the following contributions:

14

- **Boundless Memory Blocks:** It introduces the concept of using boundless memory blocks to eliminate problems (security errors, data structure corruption, premature program termination due to thrown exceptions) currently caused by fixed-size memory blocks.

- **Implementation:** It shows how to implement boundless memory blocks in a compiler that is capable of generating code for unmodified legacy C programs.

- **Evaluation:** We evaluate how well boundless memory blocks work in practice by generating versions of widely used open source server programs. Our results show that boundless memory blocks make these program invulnerable to security vulnerabilities (such as buffer overflows) caused by out of bounds memory accesses and that the overhead of using boundless memory blocks is acceptable for this set of programs.

Chapter 2

Example

We start by presenting a very simple example that illustrates how computations with boundless memory blocks operate. Figure 2-1 presents a C program that computes the sum of all positive integers found in an input file. During initialization, the program allocates an array of size 100. Then, it starts reading the input file, and whenever it finds a positive integer, it adds it to the array and updates the sum. In the end, the program prints all the integers in the array, followed by their sum.

Since the program doesn't check whether there are too many positive integers in the input file, the program will write past the end of the allocated array whenever the file contains more than 100 positive integers. While in this case the program usually terminates with a segmentation fault, an attacker may be able to fabricate a request that corrupts other potential data structures or language implementation structures, leaving the system vulnerable to executing arbitrary code contained in the malicious request.

Our boundless memory blocks compiler automatically extends the size of the array by hashing all out of bounds writes on line 20, and returning them to the corresponding out of bounds reads on lines 21 and 26. However, in order to avoid a denial of service vulnerability, by allowing an attacker to come up with a file that will consume all of the available memory, we restrict the size of the hash, by implementing it as a fixed-size LRU cache. In this case, the compiler will return a predefined value every time the program tries to read a location which is not in the cache. Note that even

in this case, our simple program will compute the sum correctly, although the actual array will contain some invalid integer values.

Every time an out of bounds read or write is detected, the compiler (optionally) writes into a log file the type of the out of bounds access (new write, overwrite, read from the cache, uninitialized read), its address in memory, its location (source file and line number), the ID of the process that generated it, and the time at which the out of bounds access was detected. This allows the developer to inspect the invalid accesses, and to correct the associated errors if desired.

```
1. #include <stdio.h>
2.
3. #define N 10
4.
5. int main() {
6.
7.    FILE *f = fopen("input.txt", "r");
8.    if (f == NULL) {
9.      printf("Error opening input file.\n");
10.     return;
11.   }
12.
13.    int *p;        // an array of integers
14.    int n = 0;    // the actual size of the array
15.    int i, aux, sum = 0;
16.    p = (int*) malloc(N*sizeof(int));
17.
18.    while (fscanf(f, "%d", &aux) != EOF)
19.      if (aux > 0) {
20.        *(p+n) = aux;
21.        sum += *(p+n);
22.        n++;
23.      }
24.
25.    for (i=0; i<n; i++)
26.      printf("Integer %d: %d\n", i+1, *(p+i));
27.
28.    printf("The sum of the integers in the input file is %d\n", sum);
29.
30.    return 0;
31. }
```

Figure 2-1: A Simple Example

We next present an example extracted from the Mutt mail client discussed in Section 4.4. Figure 2-2 presents a somewhat simplified version of a procedure from

Mutt which takes as input a string encoded in the UTF-8 format and returns as output the same string encoded in modified UTF-7 format. This conversion may increase the size of the string; the problem is that the procedure fails to allocate sufficient space in the return string for the worst-case size increase. Specifically, the procedure assumes a worst-case increase ratio of 2; the actual worst-case ratio is 7/2. When passed (the very rare) inputs with large increase ratios, the procedure attempts to write beyond the end of its output array.

With standard compilers, these writes succeed, corrupt the address space, and the program crashes with a segmentation violation. To eliminate the possibility of this kind of corruption, researchers have developed safe-C compilers that generate code that dynamically checks for and intercepts out of bounds accesses. With such compilers, Mutt exits with an out of bounds access error and does not even start the user interface. With boundless memory blocks, the program stores the additional writes away in a hash table, enabling the mail server to correctly translate the string and continue to execute correctly. Again, the developers are able to inspect the log file generated by our compiler, and to correct the errors if desired.

These examples illustrate two key aspects of using boundless memory blocks:

- **Subtle Errors:** To successfully specify a hard limit for each memory block, the programmer must reason about how all executions of the program can possibly access memory. The difficulty of performing this reason means that, in practice, real-world programs often contain subtle memory errors that can be very difficult to detect by either testing or code inspection, and these errors can have significant negative consequences for the program and its users.

- **Different Aspects of Correctness:** The fact that the programmer has failed to correctly compute the maximum possible size of the memory block or to check any possible overflows does not mean that the program as a whole is incorrect. In fact, as these examples illustrate, the rest of the computation can be completely correct once it is provided with conceptually unbounded memory blocks.

```
static char *utf8_to_utf7 (const char *u8, size_t u8len) {
    char *buf, *p;
    int ch, int n, i, b = 0, k = 0, base64 = 0;

    /* The following line allocates the return string.
       The allocated string is too small; instead of
       u8len * 2 +1, a safe length would be u8len * 4 + 1 */
    p = buf = safe_malloc (u8len * 2 + 1);

    while (u8len) {
        unsigned char c = *u8;
        if (c < 0x80) ch = c, n = 0;
        else if (c < 0xc2) goto bail;
        else if (c < 0xe0) ch = c & 0x1f, n = 1;
        else if (c < 0xf0) ch = c & 0x0f, n = 2;
        else if (c < 0xf8) ch = c & 0x07, n = 3;
        else if (c < 0xfc) ch = c & 0x03, n = 4;
        else if (c < 0xfe) ch = c & 0x01, n = 5;
        else goto bail;

        u8++, u8len--;
        if (n > u8len) goto bail;
        for (i = 0; i < n; i++) {
            if ((u8[i] & 0xc0) != 0x80) goto bail;
            ch = (ch << 6) | (u8[i] & 0x3f);
        }
        if (n > 1 && !(ch >> (n * 5 + 1))) goto bail;
        u8 += n, u8len -= n;

        if (ch < 0x20 || ch >= 0x7f) {
            if (!base64) {
                *p++ = '&';
                base64 = 1;
                b = 0;
                k = 10;
            }
            if (ch & ~0xffff) ch = 0xfffe;
            *p++ = B64Chars[b | ch >> k];
            k -= 6;
            for (; k >= 0; k -= 6)
                *p++ = B64Chars[(ch >> k) & 0x3f];
            b = (ch << (-k)) & 0x3f;
            k += 16;
        } else {
            if (base64) {
                if (k > 10) *p++ = B64Chars[b];
                *p++ = '-';
                base64 = 0;
            }
            *p++ = ch;
            if (ch == '&') *p++ = '-';
        }
    }

    if (base64) {
        if (k > 10) *p++ = B64Chars[b];
        *p++ = '-';
    }

    *p++ = '\0';
    safe_realloc ((void **) &buf, p - buf);
    return buf;

bail:
    safe_free ((void **) &buf);
    return 0;
}
```

Figure 2-2: String Encoding Conversion Procedure

Chapter 3

Implementation

Recently, several research groups have developed compilers that augment programs written in unsafe languages such as C with dynamic checks that intercept out of bounds array accesses and accesses via invalid pointers (we call such a compiler a *safe-C* compiler) [3, 37, 23, 14, 28, 15]. These checks use additional information about the (dynamic) layout of the address space to distinguish illegal accesses from legal accesses. If the program fails a dynamic check, it terminates after printing an error message.

Our implementation of boundless memory blocks for legacy C programs builds on CRED, an existing safe-C compiler [28]. The basic idea behind our implementation is to modify the generated code so that, instead of terminating the execution, it stores out of bounds writes in a hash table and implements out of bounds reads by fetching the stored values from the hash table. There are two primary issues, both of which relate to the representation of pointers:

- **Information Content:** Most safe-C compilers change the representation of pointers to enable the generated code to distinguish in bounds and out of bounds pointers [23]. Some representations use a single error token to represent all out of bounds pointers. Such representations are unsuitable for the implementation of boundless memory blocks since they do not maintain enough information to enable the generated code to identify the memory block and offset of the out

21

of bounds pointer. Our compiler therefore uses a pointer representation that maintains enough information to retrieve the memory block and offset for each out of bounds pointer.

- **Memory Layout:** Some safe-C compilers change the size of the pointer representation, which in turn changes the memory layout of the legacy C program. We decided to build on a safe-C compiler that leaves the memory layout intact, in part because this enables us to support a larger range of legacy C programs.

Our compiler generates two kinds of code: checking code and continuation code. The checking code detects out of bounds accesses; the continuation code accesses the hash table and executes when the checking code detects an out of bounds access.

3.1 Checking Code

Our implementation uses a checking scheme originally developed by Jones and Kelly [15] and then significantly enhanced by Ruwase and Lam [28]. The scheme is currently implemented as a modification to the GNU C compiler (gcc). Jones and Kelly's scheme maintains a table that maps locations to data units (each struct, array, and variable is a data unit). It uses this table to track intended data units and distinguish in-bounds from out-of-bounds pointers as follows:

- **Base Case:** A base pointer is the address of an array, struct or variable allocated on the stack or heap, or the value returned by `malloc`. All base pointers are in bounds. The *intended data unit* of the base pointer is the corresponding array, struct, variable, or allocated block of memory to which it refers.

- **Pointer Arithmetic:** All pointer arithmetic expressions contain a starting pointer (for example, a pointer variable or the name of a statically allocated array) and an offset. We say that the value of the expression is *derived from* the starting pointer. A derived pointer is in bounds if and only if the corresponding starting pointer is in bounds and the derived pointer points into the same

22

data unit as the starting pointer. Regardless of where the starting and derived pointers point, they have the same intended data unit.

- **Pointer Variables:** A pointer variable is in bounds if and only if it was assigned to an in-bounds pointer. It has the same intended data unit as the pointer to which it was assigned.

Jones and Kelly distinguish a valid out-of-bounds pointer, which points to the next byte after its intended data unit, from an invalid out-of-bounds pointer, which points to some other address not in its intended data unit. They implement this distinction by padding each data item with an extra byte. A valid out-of-bounds pointer points to this extra byte; all invalid out-of-bounds pointers have the value ILLEGAL (-2). This distinction supports code that uses valid out-of-bounds pointers in the termination condition of loops that use pointer arithmetic to scan arrays. Finally, Jones and Kelly instrument the code to check the status of each pointer before it dereferences it; attempting to dereference an out-of-bounds pointer causes the program to halt with an error.

Jones and Kelly's scheme does not support programs that first use pointer arithmetic to obtain a pointer to a location past the end of the intended data unit, then use pointer arithmetic again to jump back into the intended data unit and access data stored in this data unit. While the behavior of programs that do this is undefined according to the ANSI C standard, in practice many C programs use this technique [28]. Ruwase and Lam's extension uses an *out-of-bounds objects* (OOBs) to support such behavior [28].

As in standard C compilation, in-bounds pointers refer directly into their intended data unit. Whenever the program computes an out-of-bounds pointer, Ruwase and Lam's enhancement generates an OOB object that contains the starting address of the intended data unit and the offset from the start of that data unit. Instead of pointing off to some arbitrary memory location outside of the intended data unit or containing the value ILLEGAL (-2), the pointer points to the OOB object. The generated code checks pointer dereferences for the presence of OOB objects and uses

this mechanism to halt the program if it attempts to dereference an out-of-bounds pointer. The generated code also uses OOB objects to precisely track data unit offsets and appropriately translate pointers derived from out-of-bounds pointers back into the in-bounds pointer representation if the new pointer jumps back inside the intended data unit. In practice, this enhancement significantly increases the range of programs that can execute without terminating because of a failed memory error check [28]. This extension also has the crucial property that, unlike the Jones and Kelly scheme, it maintains enough information to determine the memory block and offset for each out of bounds pointer.

3.2 Continuation Code

Our implementation of the write continuation code stores the written value in a hash table indexed under the memory block and offset of the write. For out of bounds reads it looks up the accessed memory block and offset and returns the stored value if it is present in the hash table. If there is no indexed value, the continuation code redirects the read to a preallocated buffer of values. From the five open source applications in our test suite, only Midnight Commander contains reads that were not previously stored in the hash. For this application, we experimented with the following four patterns of preallocated values:

1. A sequence that contains only zeroes.

2. A sequence that contains zeroes on even positions and ones on odd positions.

3. A sequence of pseudo-random values in the interval from 0 to 255

4. A sequence which contains zeroes on even positions, ones on odd positions which are divisible by 3, and a sequence which iterates through all integers from 0 to 255 on the other positions.

On our standard usage workload, Midnight Commander executes successfully using any of the four patterns described above. However, since zero and one are by far

24

the most common values in computer programs [1], and since sometimes these man-
ufactured values are used to determine loop conditions (and cycling through small
integers increases the chance of having the computation hit upon a value that will
exit the loop), we decided to use the fourth pattern during our experiments.

To avoid memory leaks, it is necessary to manage the memory used to store out
of bounds writes in the hash table. Our implementation devotes a fixed amount of
memory to the hash table, in effect turning the hash table into a cache of out of
bounds writes. We use a least recently used replacement policy. It is possible for
this policy to lead to a situation in which an out of bounds read attempts to access
a discarded write entry. Our experimental results show that the distance (measured
in out of bounds memory accesses) between successive accesses to the same entry in
the hash table is relatively small and that our set of applications never attempts to
access a discarded write entry. We chose to use a fixed size cache (instead of some
other data structure that attempts to store all out of bounds writes until the program
deallocates the corresponding memory blocks) to eliminate the possibility of denial of
service attacks that cause the program to exhaust the available memory by generating
and storing a very large number of writes.

Our basic philosophy views out of bounds accesses not as errors but as normal,
although uncommon, events in the execution of the program. We acknowledge, how-
ever, that programmers may wish to be informed of out of bounds accesses so that
they can increase the size of the accessed memory block or change the program to
eliminate the out of bounds accesses. Our compiler can therefore optionally augment
the generated code to produce a log that identifies each out of bounds access. Each
entry in the log file contains the type of the out of bounds access (new write, over-
write, read from the cache, uninitialized read), its address in memory, its location
(source file and line number), the ID of the process that generated it, and the time at
which the out of bounds access was detected. Programmers can use this log to locate
and eliminate out of bounds accesses if desired.

Chapter 4

Experience

We implemented a compiler that generates code for boundless memory blocks and obtained several widely-used open-source programs with out of bounds memory accesses. Many of these programs are key components of the Linux-based open-source interactive computing environment; many of the out of bounds accesses in these programs correspond to exploitable buffer overflow security vulnerabilities.

4.1 Methodology

We evaluate the behavior of three different versions of each program: the *Standard* version compiled with a standard C compiler (this version is vulnerable to any out of bounds accesses that the program may contain), the *Check* version compiled with the CRED safe-C compiler [28] (this version terminates the program with an error message at the first out of bounds access), and the *Boundless* version compiled with our compiler (this compiler generates code to store out of bounds writes in a hash table and return the values for corresponding out of bounds reads). We evaluate three aspects of each program's behavior:

- **Security and Resilience:** We choose a workload that contains an input that triggers known out of bounds memory accesses; this input typically exploits a security vulnerability as documented by vulnerability-tracking organizations such as Security Focus [30] and SecuriTeam [29]. We observe the behavior of

the different versions on this workload, focusing on how the different programs execute after the execution of the out of bounds accesses.

- **Performance:** We chose a workload that both the Standard and Boundless versions can execute successfully (the memory errors in some programs completely disable the Check version). We use this workload to measure the *request processing time*, or the time required for each version to process representative requests. We obtain this time by instrumenting the program to record the time when it starts processing the request and the time when it stops processing the request, then subtracting the start time from the stop time.

- **Standard Usage:** When possible, we deploy the Boundless version of each program as part of our normal computational environment. During this deployment we present the program with a workload intended to simulate standard usage; we also ensure that the workload contains attacks that trigger out of bounds accesses in each program. We focus on the acceptability of the continued execution of the Boundless version of the deployed program.

We ran all the programs on a Dell workstation with two 2.8 GHz Pentium 4 processors, 2 GBytes of RAM, and running Red Hat 8.0 Linux.

4.2 Sendmail

Sendmail is the standard mail transfer agent for Linux and other Unix systems [33]. It is typically configured to run as a daemon which creates a new process to service each new mail transfer connection. This process executes a simple command language that allows the remote agent to transfer email messages to the Sendmail server, which may deliver the messages to local users or (if necessary) forward some or all of the messages on to other Sendmail servers. Versions of Sendmail earlier than 8.11.7 and 8.12.9 (8.11 and 8.12 are separate development threads) have a memory error vulnerability which is triggered when a remote attacker sends a carefully crafted email message through the Sendmail daemon [32]. When Sendmail processes the message, the memory error

causes it to execute the injected code in the message. The injected code executes with the same permissions as the Sendmail server (typically root).

4.2.1 Security and Resilience

We worked with Sendmail version 8.11.6. With Standard compilation, we were able to launch an attack that exploits the memory error to provide the attacker with a root shell. The Check version exits with a memory error during initialization and fails to operate at all. The Boundless version is not vulnerable to the attack — when sent the attack message, the process servicing the remote connection does not provide the attacker with a shell (root or otherwise). Instead, it executes through the memory error triggered by the attack to continue to successfully process subsequent Sendmail commands. We configured one version of the attack to address mail to a legitimate user; in this case the Boundless version of Sendmail delivers the message to that user (the message contains the injected code which would execute if the attack had succeeded).

4.2.2 Performance

Table 4.1 presents the request processing times for the Standard and Boundless versions of Sendmail. The Receive Small request receives a message whose body is 4 bytes long; the Send Small request sends the same message. The Receive Large request receives a message whose body is 4 KB long; the Send Large request sends the same message. We performed each request ten times and report the mean and the standard deviation of the request processing times.

Table 4.1: Request Processing Time for Sendmail (ms)

Request	Standard	Boundless	Slowdown
Receive Small	230 ± 7.47%	803 ± 4.61%	3.49
Receive Large	230 ± 19.29%	845 ± 3.06%	2.82
Send Small	23.0 ± 3.57%	88.0 ± 1.78%	3.82
Send Large	23.8 ± 3.00%	90.5 ± 0.60%	3.80

4.2.3 Standard Usage

For our standard usage workload, we installed the Boundless version of Sendmail on one of our machines and we used it to process a set of one thousand messages, composed of 960 valid messages and 40 attack messages (we sent one attack message before every 24 valid messages). On this workload, Sendmail successfully executes through the memory errors and correctly processes all the messages.

Our memory error logs indicate that Sendmail generates a steady stream of memory errors during its normal execution. In particular, every time the Sendmail daemon wakes up to check for work, it generates a flurry of memory errors. We logged 12,056 out of bounds memory accesses. All of the out of bounds read accesses retrieved values which had been previously stored in the hash table. Table 4.2 presents the different types of out of bounds accesses generated by Sendmail for our workload.

Table 4.2: Out of Bounds Accesses in Sendmail

Out of bounds accesses	12,056
Out of bounds reads[1]	12,052
Out of bounds reads from hash	12,052
Out of bounds uninitialized reads	0
Out of bounds writes[2]	4
New out of bounds writes in hash	4
Out of bounds overwrites in hash	0

4.3 Pine

Pine is a widely used mail user agent (MUA) that is distributed with the Linux operating system [27]. Pine allows users to read mail, fetch mail from an IMAP server, compose and forward mail messages, and perform other email-related tasks. We use Pine 4.44, which is distributed with Red Hat Linux version 8.0. This version of Pine has out of bounds accesses associated with a failure to correctly parse certain

---

[2]By 'reads' we denote those memory accesses which do not change the contents of the accessed memory location.

[2]By 'writes' we denote those memory accesses which change the contents of the accessed memory location.

legal From fields; by sending a user an email message with a problematic From field, it is possible for a remote attacker to exploit this vulnerability to execute arbitrary code on the user's machine [26].

4.3.1 Security and Resilience

Our security and resilience workload contains an email message with a From field that triggers this memory error. This workload causes the Standard version to corrupt its heap and abort. The Check version detects the memory error and terminates the computation with an error message identifying the error. With both of these versions, the user is unable to use Pine to read mail because Pine aborts or terminates during initialization as the mail file is loaded and before the user has a chance to interact with the program. The user must manually eliminate the From field from the mail file (using some other mail reader or file editor) before he or she can use Pine. While the Check version protects the user against injected code attacks, it prevents the user from using Pine to read mail as long as the mail file contains the problematic From field.

The Boundless version, on the other hand, continues to execute through the out of bounds accesses to enable the user to process their mail. This version processed all of our workloads without errors.

4.3.2 Performance

Figure 4.3 presents the request processing time for the Standard and Boundless versions of Pine. The Read request displays a selected email message, the Compose request brings up the user interface to compose an email message, and the Move request moves a message from one folder to another. We performed each request five times and report the mean and the standard deviation of the request processing times.

As these numbers indicate, the Boundless version is substantially slower than the Standard version for the Compose and the Move requests. However, because Pine

Table 4.3: Request Processing Time for Pine (ms)

Request	Standard	Boundless	Slowdown
Read	0.419 ± 1.88%	2.70 ± 0.55%	6.46
Compose	0.897 ± 1.35%	7.59 ± 0.41%	8.47
Move	1.44 ± 9.37%	1.86 ± 4.81%	1.29

is an interactive program, its performance is acceptable as long as it feels responsive to its users. Assuming a pause perceptibility threshold of 100 milliseconds for this kind of interactive program [6], it is clear that the application of boundless memory blocks should not degrade the program's interactive feel. Our subjective experience confirms this expectation: all pause times are imperceptible for all versions.

4.3.3 Standard Usage

For our standard usage workload, we used the Boundless version of Pine intensively for one hour to read e-mail, reply to e-mails, forward e-mails, and manage e-mail folders. To test Pine's ability to successfully execute through errors, we also periodically sent ourselves an email that triggered the memory error discussed above in Section 4.3.1.

We configured this version of Pine to generate a memory error log file; during this time we also observed several memory errors caused by errors other than the security vulnerability discussed above.

Table 4.4: Out of Bounds Accesses in Pine

Out of bounds accesses	129
Out of bounds reads	38
Out of bounds reads from hash	38
Out of bounds uninitialized reads	0
Out of bounds writes	91
New out of bounds writes in hash	91
Out of bounds overwrites in hash	0

During this usage period, the Boundless version executed successfully through all errors to perform all requests flawlessly.[3] We logged 129 out of bounds accesses. Of

---

[3]These memory errors make the Check version unusable for reading remote mail files — it exits

these out of bounds accesses, 91 modified the accessed memory location and 38 did not modify the accessed location. All of these latter 38 accesses accessed locations previously stored in the hash table. Table 4.4 presents the different types of out of bounds accesses generated by Pine for our workload.

4.4 Mutt

Mutt is a customizable, text-based mail user agent that is widely used in the Unix system administration community [22]. It is descended from ELM [10] and supports a variety of features including email threading and correct NFS mail spool locking. We used Mutt version 1.4. As described at [21] and discussed in Section 2, this version is vulnerable to an attack that exploits a memory error in the conversion from UTF-8 to UTF-7 string formats. We were able to develop an attack that exploited this vulnerability. It is possible for a remote IMAP server to use this attack to crash Mutt; it may also be possible for the IMAP server to exploit the vulnerability to inject and execute arbitrary code.

4.4.1 Security and Resilience

We configured our security and resilience workload to exploit the security vulnerability described above. On this workload, the Standard version of Mutt exits with a segmentation fault before the user interface comes up; the Check version exits with a memory error before the user interface comes up. The memory error is triggered by a carefully crafted mail folder name; when the Boundless version executes, it generates an error message indicating that the mail folder does not exist, then continues to execute to allow the user to successfully process mail from other folders.

---

whenever it attempts to access a remote mail file. It can, however, successfully process local mail files as long as they do not contain a From field that triggers the memory error.

4.4.2 Performance

Figure 4.5 presents the request processing time for the Standard and Boundless versions of Mutt. The Read request reads a selected email message and the Move request moves a message from one folder to another. We performed each request five times and report the mean request processing time.

Table 4.5: Request Processing Time for Mutt (ms)

Request	Standard	Boundless	Slowdown
Read	0.80 ± 0.78%	1.65 ± 0.87%	1.25
Move	7.40 ± 4.53%	9.29 ± 1.16%	2.06

Because Mutt is an interactive program, its performance is acceptable as long as it feels responsive to its users. These performance results make it clear that the application of boundless computing to this program should not degrade its interactive feel. Our subjective experience confirms this expectation: all pause times are imperceptible for both the Standard and Boundless versions.

Table 4.6: Out of Bounds Accesses in Mutt

Out of bounds accesses	38
Out of bounds reads	0
Out of bounds reads from hash	0
Out of bounds uninitialized reads	0
Out of bounds writes	38
New out of bounds writes in hash	38
Out of bounds overwrites in hash	0

4.4.3 Standard Usage

For our standard usage workload, we used the Boundless version of Mutt intensively for half an hour to process email messages. During this time, we triggered the security vulnerability described above twice. Mutt successfully executed through the resulting memory errors to correctly execute all of our requests. We were able to read, forward, and compose mail with no problems even after executing through the memory error.

34

An examination of the memory error log indicates that all of the memory errors were caused by the security vulnerability. We logged 38 out of bounds accesses, all of which were writes. Table 4.6 presents the different types of out of bounds accesses generated by Mutt for our workload.

4.5 Midnight Commander

Midnight Commander is an open source file management tool that allows users to browse files and archives, copy files from one folder to another, and delete files [20]. Midnight Commander is vulnerable to a memory-error attack associated with accessing an uninitialized buffer when processing symbolic links in `tgz` archives [19]. We used Midnight Commander version 4.5.55 for our experiments.

4.5.1 Security and Resilience

Our security and resilience workload contains a `tgz` archive designed to exploit this vulnerability. On this workload, the Standard version terminates with a segmentation violation when the user attempts to open the problematic `tgz` archive. Because Midnight Commander has memory errors in its initialization code, the Check version terminates with an error message before it finishes initialization. The result is that the user is unable to use this version of Midnight Commander at all.

The Boundless version, on the other hand, initializes with no problems. When the user attempts to open the problematic `tgz` archive, Midnight Commander correctly displays the names of the two symbolic links in the archive. Because these links point off to non-existent files, Midnight Commander correctly displays an error message when the user attempts to open them. Midnight Commander continues to execute successfully throughout the entire session; in particular, the user can continue to use Midnight Commander to browse, copy, or delete other files even after processing the problematic `tgz` archive. For this workload (and for all others that we know of) the application of boundless memory blocks enables Midnight Commander to exhibit completely correct behavior with no degradation at all.

4.5.2 Performance

Figure 4.7 presents the request processing time for the Standard and Boundless versions of Midnight Commander. The Copy request copies a 24Mbyte directory structure, the Move request moves a directory of the same size, the MkDir request makes a new directory, and the Delete request deletes a 3.1 Mbyte file. We performed each request five times and report the mean and the standard deviation of the request processing times.

Table 4.7: Request Processing Time for Midnight Commander (ms)

Request	Standard	Boundless	Slowdown
Copy	360 ± 0.96%	516 ± 1.28%	1.42
Move	0.451 ± 1.43 %	0.586 ± 2.02%	1.30
MkDir	0.829 ± 1.95%	1.54 ± 3.51%	1.86
Delete	2.41 ± 0.87%	2.50 ± 1.73%	1.04

As these numbers indicate, the Boundless version is not dramatically slower than the Standard version. Moreover, because Midnight Commander is an interactive program, its performance is acceptable as long as it feels responsive to its users, and these performance results make it clear that the application of boundless memory blocks to this program should not degrade its interactive feel. Our subjective experience confirms this expectation: all pause times are imperceptible for both the Standard and Boundless versions.

4.5.3 Standard Usage

For our standard usage workload, we used the Boundless version of Midnight Commander intensively for one hour. During this session, we copied, moved, browsed, and searched for files, and we created and deleted directories.

We configured this version of Midnight Commander to generate a memory error log; during this time we observed memory errors caused by coding errors other than the security vulnerability discussed above. Based on our observations, Midnight Commander generates memory errors every time it initializes, whenever it interacts

with the user through a dialog box, when it searches for a file, when it changes the current directory, and occasionally for other user interactions. The Boundless version executed successfully through all of these errors to perform flawlessly for all requests.

Table 4.8: Out of Bounds Accesses in Midnight Commander

Out of bounds accesses	16788
Out of bounds reads	16763
Out of bounds reads from hash	11301
Out of bounds uninitialized reads	5462
Out of bounds writes	25
New out of bounds writes in hash	1
Out of bounds overwrites in hash	24

During our one hour session, we logged a total of 16,788 out of bounds accesses, of which 5,462 were reads to uninitialized locations. As we will discuss in Section 4.7, Midnight Commander is the only benchmark that contains reads to locations that were not previously written by a corresponding out of bounds write. All our other benchmarks contain only reads to locations which were previously stored in the hash table. Table 4.8 presents the different types of out of bounds accesses generated by Midnight Commander for our workload.

4.6   Apache

The Apache HTTP server is the most widely used web server in the world; a recent survey found that 64% of the web sites on the Internet use Apache [24]. The Apache 2.0.47 mod_alias implementation contains a vulnerability that, under certain circumstances, allows a remote attacker to trigger a memory error [2]. The vulnerability reports indicate that this error may enable the remote attacker to inject and execute arbitrary code on the Apache server [2].

4.6.1 Security and Resilience

Our security and resilience workload contains a request that exploits the security vulnerability described above. The Apache server maintains a pool of child processes; each request is handled by a child process assigned to service the connection carrying the request [25].

With Standard compilation, the child process terminates with a segmentation violation when presented with the attack. The Apache parent process then creates a new child process to take its place. The Check version correctly processes legitimate requests without memory errors until it is presented with the attack. At this point the child process serving the connection detects the error and terminates. The parent Apache process then creates a new child process to take its place. In the Boundless version, the child process executes successfully through the attack to correctly process subsequent requests.

Because Apache isolates request processing inside a pool of regenerating processes, the Check version eliminates the security vulnerability while enabling the server to process subsequent requests. The overhead of killing and restarting child processes, however, makes this version vulnerable to an attack that ties up the server by repeatedly presenting it with requests that trigger the error.

4.6.2 Performance

Figure 4.7 presents the request processing time for the Standard and Boundless versions of Apache. The Small request serves an 8KByte page (this is the home page for our research project); the large request serves an 836KByte file used only for this experiment. We performed each request five times and report the mean and the standard deviation of the request processing times. These numbers indicate that the use of boundless memory blocks in this context entails a negligible slowdown, for both small and large requests.

Table 4.9: Request Processing Time for Apache (ms)

Request	Standard	Boundless	Slowdown
Small	180 ± 0.65%	183 ± 0.64%	1.02
Large	339 ± 0.26%	345 ± 0.14%	1.02

4.6.3 Standard Usage

For our standard usage workload, we used the Boundless version of Apache to serve the web site of our research project. For one hour, we requested files from this web site, periodically presenting the web server with requests that triggered the vulnerability discussed above. The Boundless version executed successfully through all of these attacks to continue to successfully service legitimate requests.

During our one hour session, we logged a total of 347 out of bounds accesses. All the out of bounds read accesses retrieved values which were previously stored in the hash table.

In addition to this workload, we used the Boundless version for one week to serve all requests directed to our research project's web site. This web site was in more or less steady use throughout this time period; we measured approximately 400 requests a day from outside our institution. We also generated tens of thousands of requests from another local machine, all of which were served correctly.

During this time period we periodically presented the web server with requests that triggered the vulnerability discussed above. The Boundless version executed successfully through all of these attacks to continue to successfully service legitimate requests. We observed no anomalous behavior and received no complaints from the users of the web site. Unlike some of our other programs, the memory error logs indicate that Apache had no memory errors other than those caused by our attack requests. Table 4.10 presents the different types of out of bounds accesses generated by Apache for our workload.

39

Table 4.10: Out of Bounds Accesses in Apache

Out of bounds accesses	347
Out of bounds reads	284
Out of bounds reads from hash	284
Out of bounds uninitialized reads	0
Out of bounds writes	63
New out of bounds writes in hash	32
Out of bounds overwrites in hash	31

4.7 Discussion

Our results show that boundless memory blocks enable our programs to execute through memory-error based attacks to successfully process subsequent requests. Even under very intensive workloads the Boundless versions provided completely acceptable results. We stress that we chose the programs in our study largely based on several factors: the availability of source code, the popularity of the application, the presence of known memory errors as documented on vulnerability-tracking web sites such as Security Focus [30] and SecuriTeam [29], and our ability to reproduce the documented memory errors. In all of the programs that we tested, Boundless computing successfully eliminates the negative consequences of the error — the programs were, *without exception*, invulnerable to known security attacks and able to execute through the corresponding memory errors to continue to successfully process their normal workload. These results provide encouraging evidence that the use of boundless memory blocks can go a long way towards eliminating out of bounds accesses as a source of security vulnerabilities and fatal programming errors.

One interesting aspect of our results is that although our programs generated out of bounds read accesses, in only one of these programs did any of these accesses read uninitialized values that were not previously written by a corresponding out of bounds write. This result indicates that developers are apparently more likely to incorrectly calculate a correct size for an accessed memory block (or fail to include a required bounds check) than they are to produce a program that incorrectly reads an uninitialized out of bounds memory location.

Chapter 5

Related Work

We discuss related work in the areas of memory-safe programming language implementations, traditional error recovery, and data structure repair.

5.1 Safe-C Compilers

Our work builds on previous research into implementing memory-safe versions of C [3, 37, 23, 14, 28, 15]. As described in Section 3, our implementation uses techniques originally developed by Jones and Kelly [15], then significantly refined by Ruwase and Lam [28]. Memory-safe C compilers can use a variety of techniques for detecting out of bounds memory accesses via pointers; all of these techniques modify the representation of pointers in some way as compared to standard C compilers. To implement boundless memory blocks it is essential that the pointer representation preserve the memory block and offset information for out of bounds pointers.

It is also feasible to implement boundless memory blocks for safe languages such as Java or ML by simply replacing the generated code that throws an exception in response to an out of bounds access. The new generated code, of course, would store out of bounds writes in the hash table and appropriately retrieve the stored value for out of bounds reads.

5.2 Traditional Error Recovery

The traditional error recovery mechanism is to reboot the system, with repair applied during the reboot if necessary to bring the system back up successfully [12]. Mechanisms such as fast reboots [31], checkpointing [17, 18], and partial system restarts [5] can improve the performance of the reboot process. Hardware redundancy is the standard solution for increased availability.

Boundless memory blocks differ in that they are designed to convert erroneous executions into correct executions. The advantages include better availability because of the elimination of down time and the elimination of vulnerabilities to persistent errors — restarting Pine as described in Section 4.3, for example, does not enable the user to read mail if the mail file still contains a problematic mail message.

5.3 Static Analysis and Program Annotations

A combination of static analysis and program annotations should, in principle, enable programmers to deliver programs that are completely free of out of bounds accesses. CSSV uses programmer annotations to support an analysis that can statically find all buffer overflow errors in C programs [9]. Fahndrich and Leino present an extended type system that enables the compiler to statically verify the absence of null pointer dereferences in Java programs [11]. Dhurjati, Kowshik, Adve, and Lattner present a combination of language design techniques and static analysis to statically eliminate bounds violations [8]. Xi presents a type system that ensures the absence of array bounds errors [36]. All of these techniques share the same advantage (a static guarantee that the program will not exhibit a specific kind of memory error) and drawbacks (the need for programmer annotations and the possibility of conservatively rejecting safe programs).

Researchers have also developed unsound, incomplete analyses that heuristically identify potential out of bounds memory accesses [35, 4]. The advantage is that such approaches typically require no annotations and scale better to larger programs; the

disadvantage is that (because they are unsound) they may miss some out of bounds accesses. Once again, boundless memory blocks differ from all of these techniques in that they eliminate out of bounds accesses as an error rather than attempting to detect or certify the absence of potentially out of bounds accesses.

5.4   Buffer Overflow Detection Tools

Researchers have developed techniques that are designed to detect buffer overflow attacks after they have occurred, then halt the execution of the program before the attack can take effect. StackGuard [7] and StackShield [34] modify the compiler to generate code to detect attacks that overwrite the return address on the stack; StackShield also performs range checks to detect overwritten function pointers.

It is also possible to apply buffer overflow detection directly to binaries. Purify instruments the binary to detect a range of memory errors, including out of bounds memory accesses [13]. Program shepherding uses an efficient binary interpreter to prevent an attacker from executing injected code [16].

A key difference between these techniques and boundless memory blocks is that boundless memory blocks prevent the attack from performing out of bounds writes that corrupt the address space. These writes instead are redirected into the hash table that holds the out of bounds writes. Of course, our implementation of boundless memory blocks also generates a log file that identifies all out of bounds accesses, enabling the programmer to go back and update the code to eliminate such accesses if desired.

5.5   Extensible Arrays

Many languages provide extensible array data structures, which dynamically grow to accomodate elements stored at arbitrary offsets. Boundless memory blocks are, in effect, an implementation of extensible arrays. They differ from standard extensible arrays in their tight integration with the C programming language (especially the

preservation of the address space from the original legacy implementation). This integration forces the compiler to make large scale changes to the generated code to perform the required checks and integrate effectively with the low-level packages that maintain information about out of bounds pointers and accesses.

Appendix A

Request Processing Times (All Measurements)

Table A.1: Receive Requests in Sendmail – all measurements (μs)

Receive Small		Receive Large	
Standard	Boundless	Standard	Boundless
239125	768168	267529	852687
267577	789174	296374	843536
242747	757481	325702	819378
225137	848461	460405	818175
234726	798887	268938	835721
212630	879602	259987	868751
206355	795144	312222	902899
221361	782940	254550	856501
237070	834096	285794	839855
213749	774003	267681	812343

Table A.2: Send Requests in Sendmail – all measurements (μs)

Send Small		Send Large	
Standard	Boundless	Standard	Boundless
24196	86825	23450	90162
22479	87598	23535	90601
22502	87954	23169	90435
24318	87957	25484	90050
22520	86789	22970	89831
22100	87804	23896	90048
22839	88342	23747	90907
24345	92371	23434	91831
22765	88275	24672	90648
22554	86494	23527	90667

Table A.3: Read, Compose and Move Requests in Pine – all measurements (μs)

Read Request		Compose Request		Move Request	
Standard	Boundless	Standard	Boundless	Standard	Boundless
429	2719	890	7619	1466	1748
410	2702	892	7624	1404	1936
428	2706	880	7540	1388	1943
412	2686	909	7610	1684	1761
418	2730	912	7579	1275	1932

Table A.4: Read and Move Requests in Mutt – all measurements (μs)

Read Request		Move Request	
Standard	Boundless	Standard	Boundless
802	1653	7552	9217
807	1622	7305	9283
790	1651	6809	9494
807	1662	7528	9228
803	1660	7805	9206

Table A.5: Copy and Move Requests in Midnight Commander -- all measurements (μs)

Copy Request		Move Request	
Standard	Boundless	Standard	Boundless
366698	528369	462	583
357430	509956	445	582
358492	514163	452	579
358993	511416	444	609
357721	518273	452	576

Table A.6: Mkdir and Delete Requests in Midnight Commander – all measurements (μs)

Mkdir Request		Delete Request	
Standard	Boundless	Standard	Boundless
817	1459	2420	2465
809	1504	2366	2573
830	1542	2423	2509
835	1583	2403	2480
856	1610	2416	2451

Table A.7: Small and Large Requests in Apache – all measurements (μs)

Small Request		Large Request	
Standard	Boundless	Standard	Boundless
181000	183000	340000	344000
181000	182000	338000	345000
178000	185000	340000	345000
181000	184000	339000	345000
180000	182000	338000	344000

Bibliography

[1] C. Scott Ananian and Martin Rinard. Data Size Optimizations for Java Programs. In *Proceedings of the Joint Conference on Languages, Compilers, and Tools for Embedded Systems*, June 2003.

[2] Apache HTTP Server exploit. http://securityfocus.com/bid/8911/discussion/.

[3] T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 2004.

[4] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic, programming errors. *Software - Practice and Experience*, 2000.

[5] George Candea and Armando Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 110–115, Schloss Elmau, Germany, May 2001.

[6] S. Card, T. Moran, and A. Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983.

[7] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stack-Guard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Conference*, January 1998.

[8] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, June 2003.

[9] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.

[10] ELM. http://www.instinct.org/elm/.

[11] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of OOPSLA 2003*, October 2003.

[12] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[13] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.

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

[15] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of Third International Workshop On Automatic Debugging*, May 1997.

[16] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of 11th USENIX Security Symposium*, August 2002.

[17] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, 1988.

[18] M. Litzkow and M. Solomon. The Evolution of Condor Checkpointing.

[19] Midnight Commander exploit. http://www.securityfocus.com/bid/8658/discussion/.

[20] Midnight Commander website. http://www.ibiblio.org/mc/.

[21] Mutt exploit. http://www.securiteam.com/unixfocus/5FP0T0U9FU.html.

[22] Mutt website. http://www.mutt.org.

[23] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, 2002.

[24] Netcraft website. http://news.netcraft.com/archives/web_server_survey.html.

[25] Vivek S. Pai, Peter Druschel, and Willy Zwanenepoel. Flash: An efficient and portable Web server. In *USENIX Annual Technical Conference, General Track*, 1999.

[26] Pine exploit. http://www.securityfocus.com/bid/6120/discussion.

[27] Pine website. http://www.washington.edu/pine/.

[28] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, February 2004.

[29] SecuriTeam website. http://www.securiteam.com.

[30] Security Focus website. http://www.securityfocus.com.

[31] M. I. Seltzer and C. Small. Self-monitoring and self-adapting operating systems. In *Proceedings of the Sixth workshop on Hot Topics in Operating Systems*, 1997.

[32] Sendmail exploit. http://www.securityfocus.com/bid/7230/discussion/.

[33] Sendmail website. www.sendmail.org.

[34] Stackshield. http://www.angelfire.com/sk/stackshield.

[35] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the Year 2000 Network and Distributed System Security Symposium*, 2000.

[36] Hongwei Xi and Frank Pfenning. Eliminating Array Bound Checking Through Dependent Types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.

[37] Suan Hsi Yong and Susan Horwitz. Protecting C Programs from Attacks via Invalid Pointer Dereferences. In *Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2003.