A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors)

Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, and Tudor Leu
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139

ABSTRACT

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.

We have implemented a compiler that inserts dynamic checks into the generated code to detect all out of bounds memory accesses. When it detects an out of bounds write, it stores the value away in a hash table to return as the value for corresponding out of bounds reads. The net effect is to (conceptually) give each allocated memory block unbounded size and to eliminate out of bounds accesses as a programming error.

We have acquired several widely used open source servers (Apache, Sendmail, Pine, Mutt, and Midnight Commander). With standard compilers, all of these servers are vulnerable to buffer overflow attacks as documented at security tracking web sites. Our compiler eliminates these security vulnerabilities (as well as other memory errors). Our results show that our compiler enables the servers to execute successfully through buffer overflow attacks to continue to correctly service user requests without security vulnerabilities.

1. INTRODUCTION

Standard programming languages (Fortran, C, Java, C++) allow programmers to create (statically or dynamically) 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 paper 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 default value 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 `www.securityfocus.com` and `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 paper makes the following contributions:

- **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.

2. EXAMPLE

We next present a simple example that illustrates how computations with boundless memory blocks operate. Figure 1 presents a (somewhat simplified) version of a procedure from the Mutt mail client discussed in Section 4.4. This procedure 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 7/3. 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.

This example illustrates 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 reasoning 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

```
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 1: String Encoding Conversion Procedure

maximum possible size of the memory block does not mean that the program as a whole is incorrect. In fact, as this example illustrates, the rest of the computation can be completely correct once it is provided with conceptually unbounded memory blocks.

3. IMPLEMENTATION

We have implemented boundless memory blocks for legacy C programs. Our implementation builds on an existing safe-C compiler [38]. Such compilers maintain enough information to perform (a combination of dynamic and static) checks to recognize out of bounds memory accesses. When the program attempts to perform such an access, the generated code flags the error and terminates the program. 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 [34]. 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 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 [28] and then significantly enhanced by Ruwase and Lam [38]. 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 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 [38]. Ruwase and Lam's extension uses an *out-of-bounds objects* (OOBs) to support such behavior [38].

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 [38]. 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, it returns a default value.

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 out of bounds 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, however, 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. Programmers can use this log to locate and eliminate out of bounds accesses.

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 [38] (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 chose a workload with 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 [11] and SecuriTeam [10]. We observe the behavior of the different versions on this workload, focusing on how the different programs execute after the out of bounds accesses.
- **Performance:** We chose a workload that both the Standard and Boundless versions can execute successfully. 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 use 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 [13]. 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 [12]. 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. The Standard version of Sendmail executes the out of bounds writes and corrupts its call stack. The Check version is apparently disabled by a memory error that occurs whenever the Sendmail daemon wakes up to check for incoming messages. The Boundless version is not vulnerable to the attack — it stores the out of bounds writes in the hash table and executes through the memory error triggered by the attack to continue to successfully process subsequent Sendmail commands.

4.2.2 Performance

Figure 2 presents the request processing times for the Standard and Boundless versions of Sendmail. All times are given in milliseconds. 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 Kbytes long; the Send Large request sends the same message. We performed each request at least twenty times and report the means and standard deviations of the request processing times.

Request	Standard	Boundless	Slowdown
Receive Small	15.6 ± 2.9%	72.9 ± 2.1%	4.7
Receive Large	16.8 ± 4.3%	77.9 ± 0.6%	4.6
Send Small	20.4 ± 3.3%	86.7 ± 2.4%	4.2
Send Large	21.5 ± 5.7%	88.8 ± 1.9%	4.1

Figure 2: Request Processing Times for Sendmail (milliseconds)

4.2.3 Standard Usage

For our standard usage workload, we installed the Boundless version of Sendmail on one of our machines and 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 memory error. We logged 12,017 out of bounds memory accesses. All of the out of bounds reads access values which had been previously stored in the hash table.

4.3 Pine

Pine is a widely used mail user agent (MUA) that is distributed with the Linux operating system [9]. 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 legal From fields [8].

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 3 presents the request processing times for the Standard and Boundless versions of Pine. All times are given in milliseconds. The Read request displays a selected empty message, the Compose request brings up the user interface to compose a message, and the Move request moves an empty message from one folder to another. We performed each request at least twenty times and report the means and standard deviations of the request processing times.

Request	Standard	Boundless	Slowdown
Read	0.287 ± 7.1%	2.19 ± 1.7%	7.6
Compose	0.385 ± 4.3%	3.44 ± 1.8%	8.9
Move	1.34 ± 10.4%	1.90 ± 10.0%	1.4

Figure 3: Request Processing Times for Pine (milliseconds)

As these numbers indicate, the Boundless version is substantially slower than the Standard version for the Read and Compose requests. However, because Pine 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 [18], 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. During this usage period, the Boundless version executed successfully through all errors to perform all requests flawlessly. We configured this version of Pine to generate a memory error log file. We logged 129 out of bounds accesses. Of 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.

4.4 Mutt

Mutt is a customizable, text-based mail user agent that is widely used in the Unix system administration community [6]. It is descended from ELM [2] and supports a variety of features including email threading and correct NFS mail spool locking. We used Mutt version 1.4. As described at [5] 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.

4.4.2 Performance

Figure 4 presents the request processing times for the Standard and Boundless versions of Mutt. All times are given in milliseconds. The Read request reads a selected empty message and the Move request moves an empty message from one folder to another. We performed each request at least twenty times and report the means and standard deviations of the request processing times.

Request	Standard	Boundless	Slowdown
Read	.655 ± 4.3%	2.71 ± 2.3%	4.1
Move	6.94 ± 6.2%	9.91 ± 5.9%	1.4

Figure 4: Request Processing Times for Mutt (milliseconds)

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.

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.

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.

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 [4]. Midnight Commander is vulnerable to a memory-error attack associated with accessing an uninitialized buffer when processing symbolic links in `tgz` archives [3]. 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. The Check version terminates with an error message.

The Boundless version, on the other hand, executes through the memory errors to correctly display the names of the two symbolic links in the archive. It continues on to correctly execute additional user commands; in particular, the user can continue to use Midnight Commander to browse, copy, or delete other files even after processing the problematic `tgz` archive.

4.5.2 Performance

Figure 5 presents the request processing times for the Standard and Boundless versions of Midnight Commander. All times are given in milliseconds. The Copy request copies a 31Mbyte 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 3.2 Mbyte file. We performed each request at least twenty times and report the means and standard deviations of the request processing times.

Request	Standard	Boundless	Slowdown
Copy	377 ± 0.71%	556 ± 1.54%	1.5
Move	0.30 ± 2.45 %	0.424 ± 1.69%	1.4
MkDir	0.69 ± 7.05%	1.40 ± 7.78%	2.0
Delete	2.54 ± 11.26%	2.94 ± 14.6%	1.2

Figure 5: Request Processing Times for Midnight Commander (milliseconds)

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 files, and we created and deleted directories. We periodically triggered the memory error discussed above by entering the problematic `tgz` file. We configured this version to generate an error log. This log shows that Midnight Commander has a memory error that is triggered whenever a blank line occurs in its configuration file. We verified that this error completely disabled the Check version until we removed the blank lines. The Boundless version, on the other hand, executed successfully through all memory errors to perform flawlessly for all requests.

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 of the out of bounds reads in all of our other benchmarks access locations that were previously stored in the hash table.

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 [7]. The Apache 2.0.47 mod_alias implementation contains a vulnerability that, under certain circumstances, allows a remote attacker to trigger a memory error [1].

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 [35].

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 5 presents the request processing times for the Standard and Boundless versions of Apache. All times are given in milliseconds. The Small request serves an 5KByte page (this is the home page for our research project); the large request serves an 830KByte file used only for this experiment. Both requests were local — they came from the same machine on which Apache was running. We performed each request at least twenty times and report the means and standard deviations 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.

Request	Standard	Boundless	Slowdown
Small	$44.4 \pm 1.3\%$	$46.8 \pm 1.1\%$	1.05
Large	$48.7 \pm 1.8\%$	$50.2 \pm 3.9\%$	1.03

Figure 6: Request Processing Times for Apache (milliseconds)

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 (`www.flexc.csail.mit.edu`). 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 of the out of bounds reads 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.

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 [11] and SecuriTeam [10], 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.

5. RELATED WORK

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

5.1 Memory Errors and Continued Execution

Boundless memory blocks enable the program to continue to execute through memory errors. We have also developed a technique, called *failure-oblivious computing*, which simply discards out of bounds writes and manufactures values to return as the result of out of bounds reads [36]. Even though this technique has the potential to take the program down an unanticipated execution path, in practice it enables servers to execute through memory errors (such as buffer overflows) and continue on to correctly serve subsequent requests. Another approach responds to memory errors by terminating the enclosing function and continuing on to execute the code immediately following the corresponding function call [40]. The results indicate that, in many cases, the program can continue on to execute acceptably after the premature function termination. These techniques differ from boundless memory blocks in that they are designed to convert incorrect and dangerous execution paths into unanticipated but acceptable execution paths. Boundless memory blocks, of course, can convert incorrect execution paths into correct and anticipated execution paths.

5.2 Safe-C Compilers

Our work builds on previous research into implementing memory-safe versions of C [15, 43, 34, 27, 38, 28]. As described in Section 3, our implementation uses techniques originally developed by Jones and Kelly [28], then significantly refined by Ruwase and Lam [38]. 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.

5.3 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 [23]. Mechanisms such as fast reboots [39], checkpointing [30, 31], and partial system restarts [17] 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.

5.4 Manual Error Detection and Recovery

Motivated in part by the need to avoid rebooting, researchers have developed more fine-grain error recovery mechanisms. The Lucent 5ESS switch and the IBM MVS operating system, for example, both contain software components that detect and attempt to repair inconsistent data structures [26, 33, 24]. Other techniques include failure recovery blocks and exception handlers, both of which may contain hand-coded recovery algorithms [32].

The successful application of these techniques requires the programmer to anticipate some aspects of the error and, based on this understanding, develop an appropriate recovery strategy. Boundless memory blocks, on the other hand, can be applied without programmer intervention to any system to completely eliminate memory block size calculation errors.

Data structure repair [20] occupies a middle ground. Like more traditional error detection and recovery techniques, it requires the programmer to provide some application-specific information (in the case of data structure repair, a data structure consistency specification). But because there is no explicit recovery procedure and because the consistency specification is not tied to specific blocks of code, data structure repair may enable systems to more effectively recover from unanticipated data structure corruption errors.

5.5 Static Analysis and Program Annotations

It is also possible to attack the memory error problem directly at its source: a combination of static analysis and program annotations should, in principle, enable programmers to deliver programs that are completely free of memory errors [22, 21, 42, 37]. 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 or the possibility of conservatively rejecting safe programs). Even if the analysis is not able to verify that the entire program is free of memory errors, it may be able to statically recognize some accesses that will never cause a memory error, remove the dynamic checks for those accesses, and thereby reduce the dynamic checking overhead.

Researchers have also developed unsound, incomplete analyses that heuristically identify potential errors [41, 16]. 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 genuine memory errors.

5.6 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 [19] and StackShield [14] 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 [25]. Program shepherding uses an efficient binary interpreter to prevent an attacker from executing injected code [29].

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.

5.7 Extensible Arrays

Many languages provide extensible array data structures, which dynamically grow to accommodate 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.

6. CONCLUSION

Memory errors are an important source of program failures and security vulnerabilities. This paper shows how to automatically convert legacy C programs to use (conceptually) boundless memory blocks. This conversion eliminates memory errors associated with out of bounds reads and writes and, as our results indicate, make the program invulnerable to buffer overflow attacks that exploit these errors. The measured overhead of applying our technique is acceptable for the widely used open source server programs that we tested.

Acknowledgements

This research was supported in part by the Singapore-MIT Alliance and NSF grant CCR00-86154, NSF grant CCR00-63513, NSF grant CCR00-73513, NSF grant CCR-0209075, NSF grant CCR-0341620, and NSF grant CCR-0325283.

7. REFERENCES

[1] Apache HTTP Server exploit. http://securityfocus.com/bid/8911/discussion/.
[2] ELM. http://www.instinct.org/elm/.
[3] Midnight Commander exploit. http://www.securityfocus.com/bid/8658/discussion/.
[4] Midnight Commander website. http://www.ibiblio.org/mc/.
[5] Mutt exploit. http://www.securiteam.com/unixfocus/5FP0T0U9FU.html.
[6] Mutt website. http://www.mutt.org.
[7] Netcraft website. http://news.netcraft.com/archives/web_server_survey.html.
[8] Pine exploit. http://www.securityfocus.com/bid/6120/discussion.
[9] Pine website. http://www.washington.edu/pine/.
[10] SecuriTeam website. http://www.securiteam.com.
[11] Security Focus website. http://www.securityfocus.com.
[12] Sendmail exploit. http://www.securityfocus.com/bid/7230/discussion/.
[13] Sendmail website. www.sendmail.org.
[14] Stackshield. http://www.angelfire.com/sk/stackshield.
[15] 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.
[16] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic, programming errors. *Software - Practice and Experience*, 2000.
[17] G. Candea and A. 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.
[18] S. Card, T. Moran, and A. Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983.
[19] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Conference*, January 1998.
[20] B. Demsky and M. Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2003.
[21] 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.
[22] 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.
[23] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
[24] N. Gupta, L. Jagadeesan, E. Koutsofios, and D. Weiss. Auditdraw: Generating audits the FAST way. In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, 1997.
[25] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.
[26] G. Haugk, F. Lax, R. Royer, and J. Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT&T Technical Journal*, 64(6 part 2):1385–1416, July-August 1985.
[27] 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.
[28] 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.
[29] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of 11th USENIX Security Symposium*, August 2002.
[30] 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.
[31] M. Litzkow and M. Solomon. The Evolution of Condor Checkpointing.
[32] M. R. Lyu. *Software Fault Tolerance*. John Wiley & Sons, 1995.
[33] S. Mourad and D. Andrews. On the reliability of the IBM MVS/XA operating system. *IEEE Transactions on Software Engineering*, September 1987.
[34] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, 2002.
[35] V. S. Pai, P. Druschel, and W. Zwanenepoel. Flash: An efficient and portable Web server. In *USENIX Annual Technical Conference, General Track*, 1999.
[36] M. Rinard, C. Cadar, D. Roy, D. Dumitran, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *6th Symposium on Operating Systems Design and Implementation*, Dec. 2004.
[37] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation* , June 2000.
[38] 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.
[39] 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.
[40] S. Sidiroglou, G. Giovanidis, and A. Keromytis. Using execution transactions to recover from buffer overflow attacks. Technical Report CUCS-031-04, Columbia University Computer Science Department, September 2004.
[41] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the Year 2000 Network and Distributed System Security Symposium*, 2000.
[42] H. Xi and F. Pfenning. Eliminating Array Bound Checking Through Dependent Types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.
[43] S. H. Yong and S. 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.