Failure-Oblivious Computing and Boundless Memory Blocks

Martin Rinard
MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139

***Abstract*— Memory errors are a common cause of incorrect software execution and security vulnerabilities. We have developed two new techniques that help software continue to execute successfully through memory errors: failure-oblivious computing and boundless memory blocks. The foundation of both techniques is a compiler that generates code that checks accesses via pointers to detect out of bounds accesses. Instead of terminating or throwing an exception, the generated code takes another action that keeps the program executing without memory corruption. Failure-oblivious code simply discards invalid writes and manufactures values to return for invalid reads, enabling the program to continue its normal execution path. Code that implements boundless memory blocks stores invalid writes away in a hash table to return as the values 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 implemented both techniques and 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. Both failure-oblivious computing and boundless memory blocks eliminate 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.

Key Words: Memory Errors, Buffer Overflow Attacks, Failure-Oblivious Computing, Acceptability-Oriented Computing

I. Introduction

Memory errors such as out of bounds array accesses and invalid pointer accesses are a common source of program failures. Safe languages such as ML and Java use dynamic checks to eliminate such errors — if, for example, the program attempts to access an out of bounds array element, the implementation intercepts the attempt and throws an exception. The rationale is that an invalid memory access indicates an unanticipated programming error and it is unsafe to continue the execution without first taking some action to recover from the error.

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) [15], [38], [28], [23], [33], [24]. These checks use additional information about the layout of the address space to distinguish illegal accesses from legal accesses. If the program fails a check, it terminates after printing an error message.

A. Failure-Oblivious Computing

Note that it is possible for the compiler to automatically transform the program so that, instead of throwing an exception or terminating, it simply ignores any memory errors and continues to execute normally [31]. Specifically, if the program attempts to read an out of bounds array element or use an invalid pointer to read a memory location, the implementation can simply (via any number of mechanisms) manufacture a value to supply to the program as the result of the read, and the program can continue to execute with that value. Similarly, if the program attempts to write a value to an out of bounds array element or use an invalid pointer to write a memory location, the implementation can simply discard the value and continue. We call a computation that uses this strategy a *failure-oblivious* computation, since it is oblivious to its failure to correctly access memory.

It is not immediately clear what will happen when a program uses this strategy to execute through a memory error. When we started this project, our hypothesis was that, for at least some programs, this continued execution would produce acceptable results. To test this hypothesis, we implemented a C compiler that generates failure-oblivious code, obtained some C programs with known memory errors, and observed the execution of failure-oblivious versions of these programs. We targeted memory errors in servers that correspond to security vulnerabilities as documented at vulnerability tracking web sites [11], [10]. For all of our tested servers, failure-oblivious computing 1) eliminates the security vulnerability and 2) enables the server to successfully execute through the error to continue to serve the needs of its users.

B. Boundless Memory Blocks

It is also possible to 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) [30]. 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.

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. In effect, boundless memory blocks fall back on failure-oblivious computing to avoid the possibility of unbounded memory consumption.

C. Issues

The primary difference between failure-oblivious computing and boundless memory blocks is that boundless memory blocks adhere more closely to a standard program semantics in that reads to previously written out-of-bounds locations return the previously written value. If the only error was that the programmer calculated the maximum size incorrectly, boundless memory blocks can keep the program executing with no possibility of anomalous behavior. With failure-oblivious computing, of course, the manufactured read values may cause the program to generate some unexpected results.

II. Example

We next present a simple example that illustrates how failure-oblivious computing and boundless memory blocks operate. Figure 1 presents a (somewhat simplified) version of a procedure from the Mutt mail client discussed in Section IV-D. 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 terminates with a segmentation violation. With safe-C compilers, Mutt exits with a memory error and does not even start the user interface.

A. Failure-Oblivious Computing

With our compiler that generates failure-oblivious code, the program discards all writes beyond the end of the array and the procedure returns with an incompletely translated (truncated) version of the string. Mutt then uses the return value to tell the mail server which mail folder it wants to open. The mail server responds with an error code indicating that the folder does not exist. Mutt correctly handles this error and continues

```
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;
}
```

Fig. 1. String Encoding Conversion Procedure

to execute, enabling the user to process email from other, legitimate, folders.

This example illustrates two key aspects of applying failure-oblivious computing:

- **Subtle Errors:** Real-world programs can 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.
- **Mostly Correct Programs:** Testing usually ensures that the program is mostly correct and works well except for exceptional operating conditions or inputs. Failure-oblivious computing can therefore be seen as a way to enable the program to proceed past such exceptional situations to return back within its normal operating envelope. And as this example illustrates, failure-oblivious computing can actually facilitate this return by converting unanticipated memory corruption errors into anticipated error cases that the program handles correctly.

B. Boundless Memory Blocks

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 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.

III. Implementation

We have implemented both failure-oblivious computing and boundless memory blocks for legacy C programs. Our implementation builds on an existing safe-C compiler [33]. 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 is to modify the generated code so that, instead of terminating the execution, it generates either failure-oblivious code or uses boundless memory blocks. The two primary issues are information content and memory layout. To implement boundless memory blocks, the safe-C compiler must preserve the out of bounds offsets for the accessed memory blocks. Some safe-C compilers change the size of pointers, which can change the memory layout of the C program.

Any safe-C compiler generates two kinds of code: checking code and continuation code. The checking code detects out of bounds accesses. The continuation code executes when an out of bounds access occurs. For failure-oblivious computing, the out of bounds access simply discards out of bounds writes and manufactures values for out of bounds reads. For boundless memory blocks, it stores written values in the hash table and retrieves the values for corresponding reads.

A. Checking Code

Our implementation uses a checking scheme originally developed by Jones and Kelly [24] and then significantly enhanced by Ruwase and Lam [33]. 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 [33]. Ruwase and Lam's extension uses an *out-of-bounds objects* (OOBs) to support such behavior [33].

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 [33]. 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.

B. Continuation Code for Failure-Oblivious Computing

Our implementation of the write continuation code discards the written value. The read continuation code manufactures a new value using the sequence 0, 1, 2, 0, 1, 3, 0, 1, 4, ..., wrapping around after given bound. The idea is to iterate through all small integers while returning 0 and 1 more frequently than other values since they are more frequently used in most programs. Iterating through all small integers helps programs exit loops that are looking for a given value and would otherwise loop forever.

C. Continuation Code for Boundless Memory Blocks

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.

IV. Experience

We implemented a compiler that generates, according to an input compiler flag, either code for failure-oblivious computing or code for boundless memory blocks. We also 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.

A. 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 *Failure-Oblivious Version*, which uses failure-oblivious computing, and the *Boundless Version*, which uses boundless memory blocks. 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 Failure-Oblivious and Boundless versions 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.

B. 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).

We worked with Sendmail version 8.11.6. The Standard version of Sendmail executes the out of bounds writes and corrupts its call stack. Neither the Failure-Oblivious nor the Boundless version is vulnerable to the attack — they both execute through the memory error triggered by the attack to continue to successfully process subsequent Sendmail commands. We also used both versions to process a large set of email messages, including messages that trigger the memory error. Both versions executed through the error to process subsequent messages correctly. Both the Failure-Oblivious and Boundless versions execute roughly four times slower than the Standard version [31], [30].

C. 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].

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 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.

Both the Failure-Oblivious and Boundless versions, on the other hand, continued 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, including a standard use workload that contained multiple mail messages that triggered the memory error. Our results indicate that the Failure-Oblivious and Boundless Memory Blocks version of Pine can execute up to nine times slower than the Standard version [31], [30].

D. 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 II, 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.

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 memory error is triggered by a carefully crafted mail folder name; when the Failure-Oblivious and Boundless versions execute, they generate an error message indicating that the mail folder does not exist, then continue to execute to allow the user to successfully process mail from other folders. We used both versions of Mutt for an extended user session and they both performed successfully even when presented with problematic user inputs. Our results show that the Failure-Oblivious and Boundless versions of Mutt execute approximately four times slower than the Standard version [31], [30].

E. 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.

Our security and resilience workload contains a `tgz` archive designed to exploit this vulnerability. Both the Failure-Oblivious and Boundless versions execute 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. We also used the Failure-Oblivious and Boundless versions of Midnight Commander for an extended session as part of our standard computing environment; we observed no problem with either version during this session. Our results show that the Failure-Oblivious and Boundless versions of Midnight Commander can run approximately two times slower than the Standard version [31], [30].

F. 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].

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 [29].

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. In the both the Failure-Oblivious and Boundless versions, the child process executes successfully through the attack to correctly process subsequent requests. We used both versions for extended periods of time, periodically presenting it with the attack input, and both versions executed without problems during this time. Our results show that the Failure-Oblivious and Boundless versions execute less than 10 percent slower than the Standard Version [31], [30].

G. Discussion

Our results show that both failure-oblivious computing and boundless memory blocks enable our programs to execute through memory-error based attacks to successfully process subsequent requests. Even under very intensive workloads the both 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, both versions successfully eliminated 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 failure-oblivious computing or 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.

V. 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.

A. Memory Errors and Continued Execution

Boundless memory blocks enable the program to continue to execute through memory errors. Another approach responds to memory errors by terminating the enclosing function and continuing on to execute the code immediately following the corresponding function call [35]. The results indicate that, in many cases, the program can continue on to execute acceptably after the premature function termination.

B. Extensible Arrays

Extensible array data structures, which dynamically grow to accommodate elements stored at arbitrary offsets, are a known technique in computer science. 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.

C. Safe-C Compilers

Our work builds on previous research into implementing memory-safe versions of C [15], [38], [28], [23], [33], [24]. As described in Section III, our implementation uses techniques originally developed by Jones and Kelly [24], then significantly refined by Ruwase and Lam [33]. 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 generated code would store out of bounds writes in the hash table and appropriately retrieve the stored value for out of bounds reads.

D. 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 [21]. Mechanisms such as fast reboots [34], checkpointing [26], [27], 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 IV-C, for example, does not enable the user to read mail if the mail file still contains a problematic mail message.

E. 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 [20], [19], [37], [32]. 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 [36], [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.

F. 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 [18] 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 [22]. Program shepherding uses an efficient binary interpreter to prevent an attacker from executing injected code [25].

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.

VI. Conclusion

Memory errors are a serious problem in software systems today, leading to unanticipated and undesirable program execution and potentially even causing security violations. We have presented two techniques, Failure-Oblivous Computing and Boundless Memory Blocks, that have the potential to ameliorate many of the negative effects of memory errors. Both techniques use bounds checks to prevent data corruption. They both continue through the memory error to allow the program to continue to execute. We have implemented both techniques; our results show that they can enable programs to continue to execute successfully and continue to service their legitimate users even after the programs are presented with inputs that trigger memory errors.

Acknowledgements

The research presented in this paper was performed with Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. It was supported in part by the Singapore-MIT alliance, DARPA Cooperative Agreement FA 8750-04-2-0254, DARPA Contract 33615-00-C-1692, and NSF Grants CCR-0341620, CCR-0325283, and CCR-0086154.

References

[1] Apache HTTP Server exploit. www.securityfocus.com/bid/8911/discussion/.
[2] ELM. www.instinct.org/elm/.
[3] Midnight Commander exploit. www.securityfocus.com/bid/8658/discussion/.
[4] Midnight Commander website. www.ibiblio.org/mc/.
[5] Mutt exploit. www.securiteam.com/unixfocus/5FP0T0U9FU.html.
[6] Mutt website. www.mutt.org.
[7] Netcraft website. http://news.netcraft.com/archives/web_server_survey.html.
[8] Pine exploit. www.securityfocus.com/bid/6120/discussion.
[9] Pine website. www.washington.edu/pine/.
[10] SecuriTeam website. www.securiteam.com.
[11] Security Focus website. www.securityfocus.com.
[12] Sendmail exploit. www.securityfocus.com/bid/7230/discussion/.
[13] Sendmail website. www.sendmail.org.
[14] Stackshield. 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] 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.
[18] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Conference*, January 1998.
[19] 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.

[20] 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.

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

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

[23] 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.

[24] 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.

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

[26] 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.

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

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

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

[30] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, , and Tudor Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *Proceedings of the 2004 Annual Computer Security Applications Conference* , December 2004.

[31] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, , and Jr. William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* , December 2004.

[32] Radu Rugina and Martin 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.