A Practical Dynamic Buffer Overflow Detector

Olatunji Ruwase

Transmeta Corporation
3990 Freedom Circle
Santa Clara, CA 95054
tjruwase@transmeta.com

Monica S. Lam

Computer Systems Laboratory
Stanford University
Stanford, CA 94305
lam@stanford.edu

Abstract

Despite previous efforts in auditing software manually and automatically, buffer overruns are still being discovered in programs in use. A dynamic bounds checker detects buffer overruns in erroneous software before it occurs and thereby prevents attacks from corrupting the integrity of the system.

Dynamic buffer overrun detectors have not been adopted widely because they either (1) cannot guard against all buffer overrun attacks, (2) break existing code, or (3) incur too high an overhead. This paper presents a practical detector called CRED (C Range Error Detector) that avoids each of these deficiencies. CRED finds all buffer overrun attacks as it directly checks for the bounds of memory accesses. Unlike the original referent-object based bounds-checking technique, CRED does not break existing code because it uses a novel solution to support program manipulation of out-of-bounds addresses. Finally, by restricting the bounds checks to strings in a program, CRED's overhead is greatly reduced without sacrificing protection in the experiments we performed.

CRED is implemented as an extension of the GNU C compiler version 3.3.1. The simplicity of our design makes possible a robust implementation that has been tested on over 20 open-source programs, comprising over 1.2 million lines of C code. CRED proved effective in detecting buffer overrun attacks on programs with known vulnerabilities, and is the only tool found to guard against a testbed of 20 different buffer overflow attacks[34]. Finding overruns only on strings impose an overhead of less than 26% for 14 of the programs, and an overhead of up to 130% for the remaining six, while the previous state-of-the-art bounds checker by Jones and Kelly breaks 60% of the programs and is 12 times slower. Incorporating well-known techniques for optimizing bounds checking into CRED could lead to further performance improvements.

This research was performed while the first author was at Stanford University, and this material is based upon work supported in part by the National Science Foundation under Grant No. 0086160.

1. Introduction

Buffer overflows are the most common form of security threat in software systems today, and vulnerabilities attributed to buffer overflows have consistently dominated CERT advisories[7]. In the year 2002, 57% of security advisories for the year were related to buffer overflow vulnerabilities. As of August 2003, 50% of the security advisories issued for the year fell under this category. In addition, 50% of the 60 most severe vulnerabilities as posted on CERT/CC were caused by buffer overflow errors in programs[8]. A similar pattern is also observable in vulnerabilities listings posted on computer security websites, such as SecurityFocus[27] and Securiteam[26]. Computer worms such as Slammer, CodeRed, and more recently, Blaster and Welchia have exploited buffer overflow vulnerabilities in programs to inflict billions of dollars worth of damages on the computing community. An effective solution to buffer overruns will have a significant impact in improving the security of our computing systems.

1.1. Background

In a buffer-overflow attack on a vulnerable program, the attacker attempts to modify the memory state of the program so that it yields control of the machine to the attacker in the privilege mode of the program. To launch an attack, the attacker would supply carefully-crafted excess

input data to the program. A program that does not check if the input exceeds its memory buffer size would copy the excess data into locations adjacent to the buffer. By controlling the contents, the attacker can cause the program to deviate from its intended purpose.

A classic and simple example of such an attack, known as stack smashing[1], simply overwrites the return address of a function on the stack so that, when the function returns, control jumps to a location where the attacker would have inserted malicious code. Other more complex variants attempt to modify locations referenced by function pointers and the global offset table.

Although the buffer overrun problem surfaced many years ago, a practical solution has eluded the software community despite efforts in code auditing and development of static and dynamic buffer overflow detectors. Static buffer overflow detectors attempt to verify that all memory accesses are checked for overruns[11, 22, 31]. The problem, unfortunately, is undecidable in general. Sound tools tend to generate too many false warnings and unsound tools can miss errors in the code. Moreover, warnings generated by the analysis tools require that programmers manually inspect the code and insert in the appropriate checks. These weaknesses render static buffer overflow detectors impractical.

Dynamic buffer overflow detectors are attractive because they automatically insert the necessary guards. However, for a dynamic detector to be deployed: it must (1) protect against all buffer overflow attacks, (2) not break working code, and (3) be reasonably efficient. There are no buffer overflow detectors proposed to date that satisfy all these requirements.

Some dynamic buffer overflow detectors do not offer complete protection against buffer overflow attacks; tools such as StackGuard[10], StackShield[30], and Propolice[12] attempt to guard against only stack smashing. Bounds checkers detect any bounds violations in program execution and hence guard against all buffer overflow attacks. Some bounds checkers modify the representation of C pointers[9, 17, 19]. They replace each pointer in the program with a structure that holds information needed for bounds checking, such as the base and extent address of the buffer referenced by the pointer. Not only does such modification increase memory storage significantly, it is incompatible with legacy code.

The bounds checker proposed by Jones and Kelly is particularly attractive in that no pointer representation modifications are necessary[18]. They define the *referent object* of a pointer to be the program buffer that the pointer is intended to reference. They observe that the result of a pointer arithmetic operation must have the same referent object as the original pointer. Thus, given an in-bounds pointer, and the ability to retrieve the base and extent of the referent object, the checker can determine if the computed address is in-bounds. However, their approach cannot handle the case where an out-of-bounds pointer to an object is stored and later retrieved to compute an in-bounds address. This weakness causes the tool to generate false alarms on many existing programs. Furthermore this tool imposes significant runtime overhead because all non copy pointer operations are instrumented.

1.2. Contributions

This paper presents a practical dynamic buffer overrun detector called CRED (C Range Error Detector). CRED enforces a relaxed standard of correctness by allowing program manipulations of out-of-bounds addresses (a violation of the ANSI C standard) that do not result in buffer overflows. The idea is to replace every out-of-bounds pointer value with the address of a special OOB (out-of-bounds) object created for that value. Kept with the OOB object is the actual pointer value and information on the referent object. The OOB addresses can be copied around arbitrarily, just like any other data. When the value is used as an address, however, it is replaced by the actual out-of-bounds address. Any pointer derived from the address is bounds checked before it can be dereferenced. Our design supports the relatively rare out-of-bounds address manipulation, without increasing the overhead for the common case. Also, it minimizes the space overhead by reclaiming storage associated with deallocated pointers, ensuring that no memory is leaked. In addition, CRED exploits type information to minimize run-time overhead by restricting checks to only accesses that present security risks.

We have implemented our idea on top of Jones and Kelly's extension to the GNU C compiler version 3.3.1. We tested our implementation extensively by running it on 20 open-source programs, which together comprise over 1.2 million lines of code. Unlike Jones and Kelly's implementation, which failed on 60% of the programs, code generated by our compiler passed all the test suites. Our system is also shown to be effective in detecting buffer overflows. It is the only tool reported so far to be effective against a testbed of 20 different buffer overflow attacks[34]. Our optimizations impose an overhead of less than 26% for 14 of the programs, and an overhead of up to 130% for the remaining six, while the previous state-of-the-art tool by Jones and Kelly runs 12 times slower.

This research shows that it is feasible to build a simple, robust, and compatible C bounds checker that can guard against all buffer overruns. The performance of our system can be further improved by applying known compiler

optimization techniques.

1.3. Paper Organization

We present the bounds-checking technique based on referent objects in Section 2, discuss its deficiency, and describe our proposed solution. We then discuss an optimization in Section 3. Section 4 presents experimental results of applying CRED to commonly used software applications. In Section 5, we review other approaches to tackling the buffer overflow problem. Section 6 presents our conclusion and future work.

2. Bounds Checking Using Referent Objects

To check the bounds of a pointer, we need to determine its referent object, and then check if the pointer is within bounds. Because C is not type safe, allows arbitrary address calculations, and allows pointers to point to the middle of an object, it is not easy to determine the referent object of a pointer.

The referent-object based approach proposed by Jones and Kelly[18] is based on the principle that an address computed from an in-bounds pointer must share the same referent object as the original pointer.

Their scheme uses an *object table*, a run-time data structure that collects all the base address and size information of all static, heap and stack objects. To determine if an address computed off an in-bounds pointer is in-bounds, the checker first locates the referent object by comparing the in-bounds pointer with the base and size information stored in the object table. Then, it checks if the new address falls within the extent of the referent object.

The object table is implemented as a splay tree[29] to optimize the pointer lookups. The size of the object table is relatively small because it is linear with respect to the number of objects, and not the number of pointers, during program execution.

Jones and Kelly's tool was implemented as a run-time library of checking routines. They modified the front end of gcc so that all object creation, address manipulations and dereference operations are intercepted and replaced with calls to appropriate routines in the checking library. These calls make sure that the object table is kept up to date and that all the addresses generated and dereferenced are within bounds. In addition, the tool provides a bounds-checked version of each of the unsafe standard C library routines that have been instrumental to successful exploits of vulnerable programs.

An important advantage of Jones and Kelly's approach is that code instrumented with their tool is compatible with uninstrumented code, such as linked libraries. Clearly, since the representation of pointers has not been changed, uninstrumented codes can work with pointers and objects created by the instrumented code without any modification. Conversely, it is also easy to allow instrumented codes to work with pointers pointing to objects created by uninstrumented codes.

Heap objects created by uninstrumented codes are checked like all other heap objects. They are allocated and de-allocated using bounds-checked versions of malloc and free that appropriately update the object table. Stack and static objects declared in uninstrumented code are called *unchecked objects* and do not appear in the object table. Unchecked stack objects are located between the stack pointer and the top of the stack. Unchecked static objects are located between address 0 of the virtual memory and the one byte beyond the _BSS section, or in the region where dynamically linked libraries are loaded. Thus, a bounds checker can easily determine if an address points to an unchecked object and not bother checking its bounds.

2.1. Jones and Kelly's Design for Handling Out-of-bounds Pointers

The design described so far assumes that every address is computed from a pointer that is known to be in-bounds. What if we compute an out-of-bounds address, store it, and then access it later? How do we know it is out of bounds? Jones and Kelly's design provides an answer to this question. However, their design breaks if a stored out-of-bounds address is used subsequently to compute an in-bounds address. This, unfortunately, happens quite often.

The C standard offers one possible way to the handle out-of-bounds pointers. In the C standard, pointer arithmetic is well defined if and only if the resulting address lies within the extent of the array, or if it points to the very next byte after the array. This rule also applies to scalar objects by treating them simply as an array of one element. The C standard allows the generation of the address pointing immediately past an array because the value is often used to determine if the end of the array is reached in a loop. While it is legal to generate the address immediately past an array, dereferencing it results in undefined behavior. For all other out-of-bounds addresses, both the generation of the address and the dereference operations are considered to be undefined.

Jones and Kelly's approach takes full advantage of the language standard in their design. First, to handle the off-by-one addresses, they pad all objects, with the exception of parameters to a function, by an extra byte. This means that a pointer pointing immediately past an object can be

easily identified. The pointer's intended referent is simply the object preceding the address; the pointer can be compared with other pointers to the same object, it can also be stored, but dereferencing it would cause a buffer overrun. Parameters in a function are not padded, otherwise instrumented and uninstrumented code would have different parameter layouts.

Second, all other out-of-bounds addresses resulting from pointer arithmetic operations are replaced with a special ILLEGAL value, defined as `(void*)` -2. These addresses can be copied without requiring any special handling. They can be cast to other types, as a pointer to an array of characters for instance, before they are copied. The ILLEGAL value is not allowed to be dereferenced or used to generate an address. Any such operation, easily identifiable because it is preceded by a cast to a pointer, will cause the run-time system to halt the program and report a buffer overrun error.

Unfortunately, many existing programs do not follow the C standard; 60% of the programs we tested fell into this category. In particular, there are programs that generate and store out-of-bound addresses and later retrieve these values in their computation, without causing buffer overruns. For example, they may be used in comparisons and computations of in-bounds addresses. Jones and Kelly's approach substitutes the out-of-bound address with a special ILLEGAL value whenever it is computed, thus causing such programs to no longer behave the same.

Figure 1(a) shows a simple C program that demonstrates this problem. This program only allocates a heap buffer to pointer `p` and sets all other pointers to in-bounds and out-of-bounds locations of the buffer. Figure 1(b) shows the memory state of an uninstrumented code after line 6. While `s` points to an illegal address, `r` points to a legal one and can be dereferenced without causing an error. If the code has been instrumented, the memory state after line 5 is shown in Figure 1(c). `p`, `q` have the same referent object. `s` is found to be out of bounds and is therefore set to the ILLEGAL value (-2). The checker incorrectly crashes the program at line 6, since it does not permit arithmetic on out-of-bounds pointers. It is unacceptable for a checker to break working code. Figure 1(d), described in more detail below, shows how our proposed technique handles this program.

2.2. Proposed Out-of-Bound Addresses Handling

As discussed above, it is unacceptable to lose the stored value of an out-of-bounds address. We must retain the value of the pointer, and at the same time, keep track of its referent object. Our approach is to create a unique *out-of-bounds object* (OOB object) in the heap for every stored out-of-bounds address value, and substitute the value with the address of the corresponding OOB object. These objects are deallocated as soon as they are no longer needed to minimize the space overhead. An OOB object contains (a) the out-of-bounds address value and (b) the referent object that the value refers to. It is not entered into the object table, but rather entered into an *out-of-bounds object hash table* (OOB hash table).

We can check if a pointer points to an OOB object quickly by consulting this hash table. The hash table is consulted only in the rare case where the checker could neither find the referent object of a pointer used in a non-copy operation in the object table nor identify the object as unchecked.

Let us now describe our technique step by step:

1. After every address computation, the run-time system checks if the address is out-of-bounds. If so, a special `malloc` is invoked to create an OOB object; the OOB object is not recorded as a regular object in the object table, but its address is entered into the OOB hash table. The out-of-bounds address and the referent object's address are stored in the OOB object.

2. When a pointer is dereferenced, check if it points to an object in the object table or to an unchecked object. If neither is the case, it is an illegal reference and the program is halted after an appropriate error message is printed.

3. If a pointer is used in an arithmetic or comparison operation, checks if it points to an object or to an unchecked object. If neither is the case, check the OOB hash table to determine if it is an out-of-bounds value. The referent object and its value are retrieved from the OOB itself. The desired operation is performed on the actual out-of-bounds value.

4. When an object is de-allocated, implicitly if it is on the stack and explicitly if it is on the heap, delete all OOB objects referring to the object. This prevents the hash table and the number of OOB objects from growing indefinitely. Simply scan the hash table for any OOB object whose referent object is being deleted, and delete the OOB object as well as the heap entry.

Figure 1(d) demonstrates how this technique would work on our example program. It shows the memory state of a CRED instrumented executable after line 6. The OOB object allows us to correctly determine the referent object

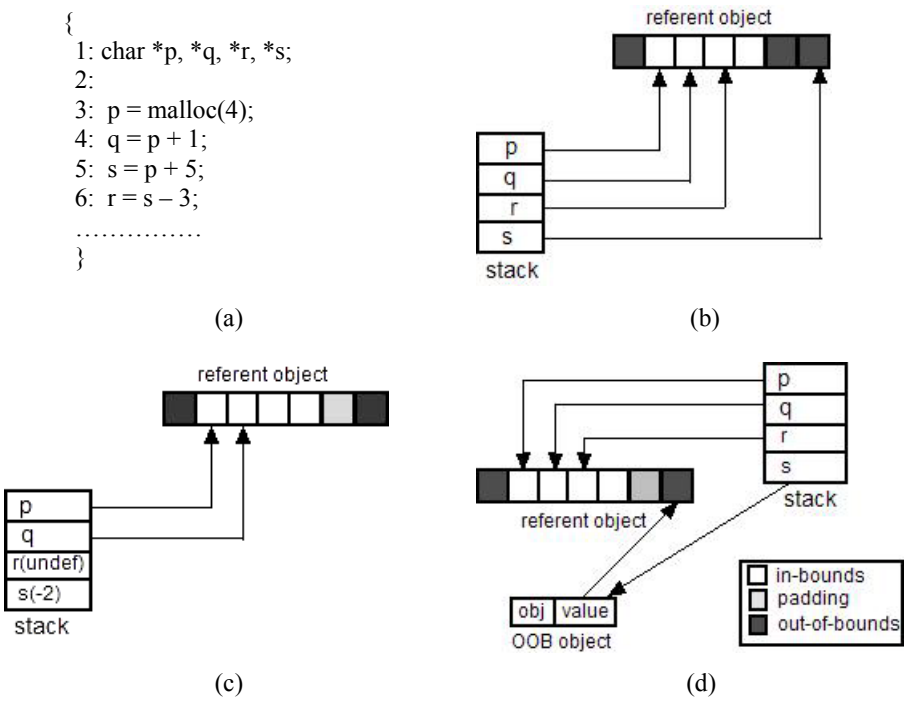

Figure 1. (a) Simple C program, memory states of (b) uninstrumented execution, (c) instrumentation with Jones and Kelly Checker, (d) instrumentation with CRED.

and value of `r` from the arithmetic operation on `s`. CRED is therefore compatible with real-life programs.

Note that there are situations where our scheme would fail. Assume an out-of-bounds pointer is cast to an integer, used in arithmetic operation, and the result cast back to a pointer. Subsequent use of the resulting pointer could lead to undetected memory safety violations since it may reference an object in the object table. One way to guard against such problems is to perform an analysis over the program to locate and warn of any unsafe cast operations. Furthermore, similar to the Jones and Kelly checker, CRED's interaction with external libraries requires no special handling for passing in-bounds pointers. However, for out-of-bounds pointers, the address of the OOB object would be incorrectly passed resulting in undefined behavior if the value is used in a non copy operation. We believe this case is rare in correct code. Note that we have not encountered such cases in the million lines of code we tested.

3. Run-time Overhead

The poor performance of the original Jones and Kelly's technique is yet another obstacle to its adoption. This scheme incurs a run-time overhead on every memory access involving arrays and pointers. Virtually all large, useful software uses pointers extensively and thus suffers from a significant performance degradation.

We recognize that buffer overflow attacks are perpetrated by overflowing a program buffer using user-supplied string data. Thus, for security purposes, we only need to check the bounds of string data and thus significantly reduce the overhead of securing software systems, without compromising the quality of protection offered.

We modified the parsing files of gcc to check the bounds of only character arrays and pointers during the construction of the abstract syntax tree. We introduced a compilation flag for specifying compilation in this mode. Thus, at run time, calls to the bounds-checking library are made only for strings. We maintain the table for all objects regardless of type so that the bounds-checking versions of the library functions can handle casts correctly. It is important to observe that when data is copied be-

Program	Type	# Lines	Vuln.	Tests	JK	CRED
Apache-1.3.24	web server	73.6K	no	yes	fail	pass
binutils-2.13.2.1	binary tools	596.5K	no	yes	fail	pass
bison-1.875	parser generator	25.1K	no	yes	fail	pass
ccrypt-1.4	encryption utility	4.4K	no	yes	pass	pass
coreutils-5.0	file, shell, & text utilities	69.5K	no	yes	fail	pass
enscript-1.6.1	ascii to postscript converter	22.1K	no	yes	fail	pass
gawk-3.1.2	string manipulation tool	36.4K	yes	yes	fail	pass
gnupg-1.2.2	OpenPGP implementation	71.2K	no	yes	fail	pass
grep-2.5.1	pattern matching utility	20.8K	no	yes	fail	pass
gzip-1.2.4	compression utility	5.8K	yes	yes	pass	pass
hypermail-2.1.5	mail to HTML converter	27.6K	yes	yes	fail	pass
monkey-0.7.1	web server	2.5K	yes	no	pass	pass
OpenSSH-3.2.2p1	SSH1 protocol implementation	43.4K	no	no	fail	pass
OpenSSL-0.9.7b	SSL & TLS toolkit	162.7K	no	yes	fail	pass
pgp4pine-1.76	mail encryption tool	3.3K	yes	no	fail	pass
polymorph-0.40	filesystem unixier	0.4K	yes	no	pass	pass
tar-1.13	archiving utility	18.2K	no	yes	pass	pass
WsMp3-0.0.10	web server	3.4K	yes	no	pass	pass
wu-ftpd-2.6.1	FTP server	18.3K	no	no	pass	pass
zlib-1.13	data compression library	8.3K	no	yes	pass	pass

Figure 2. Results of compatibility experiment.

tween locations in memory, objects are usually cast down to character pointers and not vice versa. This observation is important in appreciating that security is not compromised by this technique.

4. Experiments

We have implemented the techniques described above in a tool called CRED. CRED has been merged into the latest Jones and Kelly checker for gcc 3.3.1, which is currently maintained by Brugge[5]. We refer to the Jones and Kelly checker as JK in this section.

We carried out experiments to evaluate the effectiveness of CRED with respect to correctness, protection offered against buffer overflows and the performance improvements obtained by strings-only checking. Here we first describe the applications then our experimental results.

4.1. Application Programs

We used some common open-source programs for this evaluation. Figure 2 shows the list of programs, along with the lines of code count as generated using SLOCCount[32]. The "Tests" column indicates whether the program source was distributed with a test suite. The column labeled "Vuln" indicates that the program had a known vulnerability and a publicly available exploit. Seven of the programs fell into this category and were used to evaluate the protection offered by CRED. More than 1.2M lines of C code was evaluated during this experiment.

4.2. Compatibility

We first evaluated the compatibility of CRED with real-life programs and compared it with that of JK. As a stress test, we ran this experiment by checking for overflows for all buffers, and not just strings. Most open-source programs are distributed with a suite of self tests. We took advantage of the test suites for the experiment whenever they were available, and used simple tests for those programs without a test suite.

The last two columns in Figure 2 indicate the result of this experiment. The Jones and Kelly extension failed on 12 out of the 20 programs. This suggests that most of the programs violate the C standard by manipulating out-of-bounds addresses that do not just point immediately past an object. CRED, in full bounds-checking mode, passed all the tests. This experiment also uncovered a number of previously unknown bounds errors in non-string buffers when executing the test suites. This finding led to bug

fixes in coreutils, bison and OpenSSL. This result suggests that CRED, in its full bounds checking mode, could be a useful for software development and testing.

4.3. Protection

We evaluated the effectiveness of CRED in protecting vulnerable programs against buffer overflow attacks with two experiments. These experiments were performed with the optimization of checking only for overruns in strings.

Our first experiment was carried out using the seven vulnerable programs described earlier in Section 4.1. Each program was instrumented with CRED and attacks were launched on them. In each case, the attempted overflow was detected and the program halted with an appropriate error message.

The second experiment tested the effectiveness of CRED on a testbed of 20 different buffer overflow attacks developed by Wilander and Kamkar for evaluating dynamic buffer overflow detectors[34]. The implemented attacks used two overflow techniques. These are either to overflow the buffer all the way to the target or to redirect a pointer to the target. The targets are the return address, function pointers, old base pointer and longjmp buffers. The overflows are attempted on the stack, heap, bss and data segments. ProPolice[12], StackGuard[10], StackShield[30], and Libsafe and Libverify[2] were evaluated in the report. CRED successfully detected all of the attacks in the testbed. ProPolice[12], the best of the tools evaluated by Wilander and Kamkar[34], could only detect 50% of the attacks in the testbed. These experiments demonstrate that restricting bounds checking to strings only is effective in thwarting buffer overflow attacks on vulnerable programs.

4.4. Performance

CRED offers better backward compatibility than the JK technique because it tracks out-of-bounds address values. To determine the performance overhead of tracking out-of-bounds addresses, we compare the performance of CRED, in its full bounds-checking mode, with JK. Only the 8 programs with which JK is compatible were used for this experiment. The experiment was conducted by measuring execution times it took the instrumented executable to run the test suite. For programs without a test suite, we ran simple tests described in Figure 4. The results are presented in Figure 3. The results indicate that the worst relative performance experienced is a 15% slowdown in tar. The differences are negligible in all other cases. We also evaluated the space used to maintain OOB data structures.

	JK(s)	CRED (s)
ccrypt	26.93	23.00
gzip-1.2.4	0.19	0.18
monkey-0.7.1	5.60	6.00
polymorh-0.4.0	0.39	0.39
tar-1.13	0.66	0.76
WsMp3-0.0.10	1.48	1.48
wu-ftpd-2.6.1	33.40	33.40
zlib-1.13	0.11	0.11

Figure 3. Execution times of JK **and** CRED **(full bounds-checking mode) instrumentation for programs that are compatible with** JK**.**

The largest utilization observed at any given point in time was 976 bytes for CRED-instrumented `bison`. This low overhead was not surprising as it is unlikely that a correct program would generate many out-of-bounds pointers at any given point in time.

The second experiment evaluated the performance improvements resulting from strings-only checking for all of the test programs. Except for those applications noted in Figure 4, we evaluated the performance of the system by timing the execution of the test suites. Apache and OpenSSL are evaluated with their standard benchmarking tools, httperf[25] for Apache and speed for openssl. The experiments were carried out on a 2.4Ghz Intel Pentium 4, 1GB Linux system using gcc-3.0.4 compiler with -O2 optimization level. This machine also ran the server in experiments involving the Apache, monkey and WsMp3 servers, while the client ran on a 600Mhz, dual CPU, Pentium III, 1GB Linux machine.

The results of the experiment are presented in Figure 5. Full bounds checking, like the original JK system, imposes significant performance overhead on most programs. The enscript application experiences a 11-times slowdown, while ssh experiences a 12-times slowdown when instrumented with full bounds checking. Limiting the bounds-checking to strings greatly improves the performance for most programs. The instrumentation imposes an overhead of less than 26% for 14 of the programs. The slowdown is still significant for Apache (1.6X), enscript (1.9X), gnupg (1.8X), hypermail (2.3X), monkey (1.8X) and pgp4pine (1.6X). These programs involve significant string processing, thereby limiting the effectiveness of our optimization. Fortunately, known compiler optimizations can be applied to eliminate redun-

Program	What was evaluated
Apache-1.3.24	Response time to 15K tcp connections at the rate of 90 per second.
monkey-0.7.1	Response time to 3K tcp connections at the rate of 50 per second.
openssh-3.2.2p1	Latency of 126MB file transfer using scp via the network loop back interface.
OpenSSL-0.9.7b	Time to sign and verify 2048 bit keys using rsa.
pgp4pine-1.76	Time to decrypt 1MB file
polymorph-0.40	Time to convert names of 100 files to unix style (lower case) names.
WsMp3-0.0.10	Latency of downloading a 1.5MB file.
wu-ftpd-2.6.1	Latency of 126MB file transfer via the network loop back interface

Figure 4. Description of performance tests.

dant bounds checks in loops and thereby reduce the performance penalty[15]. Static analysis techniques can also be incorporated to reduce the portion of code that requires instrumentation[9].

5. Related Work

A considerable amount of work has been performed on mitigating the buffer overflow problem using either static analysis or dynamic analysis. In addition comparative studies of these techniques have been carried out[14, 28, 33, 34]. In this section we review different works in each category.

5.1. Static Analysis Approaches

Wagner et al. developed a system for detecting buffer overflows in C programs statically[31]. Their approach treats C strings as an abstract data type accessed through the library routines and models buffers as pairs of integer ranges (size and current length), while the detection problem is formulated as an integer constraint problem. The library functions are modeled in terms of how they modify the size and length of strings. By trading precision for scalability their implemented tool gives both false positives and false negatives. The tool found known and unknown security vulnerabilities in Sendmail 8.7.5.

Larochelle and Evans[22] presented a lightweight annotation-assisted static analysis based on LCLint[13]. This technique exploits information provided in programmers' semantic comments to detect likely buffer overflow vulnerabilities. Their tool is neither sound nor complete.

Sagiv et al. presented C String Static Verifyer (CSSV), a tool that statically detects string manipulation errors with the aid of procedure summaries[11]. CSSV is sound and handles all C language constructs such as structures and multilevel pointers. Its disadvantages are that it generates false alarms and imposes on the programmer the extra burden of writing procedure summaries.

5.2. Dynamic Analysis Approaches

StackGuard by Cowan et al. is an extension to the GNU C compiler that tackles the stack smashing attacks by inserting a canary word just before the return address on the stack[10]. Attempts to overwrite the return address would result in the canary value being modified. The canary is verified when the function is about to return, and the program is halted if the canary was altered. Bulba and Kil3r present techniques for bypassing StackGuard[6]. Our tool is not susceptible to these techniques.

ProPolice by Etoh and Yoda is another extension to the GNU C compiler that protects against stack smashing attacks [12]. Similar to StackGuard, ProPolice protects the return address with a guard value. In addition stack allocated variables are rearranged such that local char buffers are at a higher address(below the guard value) than other local variables and pointers. Therefore local char buffers cannot be overflowed to affect other local variables. ProPolice offers no protection against other forms of buffer overflow attacks, which our tool does.

StackShield is also an extension to the GNU C compiler that protects the return address against stack smashing attacks[30]. It does so by storing a copy of the return address in a safe place on entering a function and restoring it before returning. So if the return address on the stack is overwritten, the saved copy will be restored anyway and used when the function returns. Techniques to bypass StackShield are presented by Bulba and Kil3r[6]. Our tool is impervious to these techniques and thus offers better protection to programs.

Baratloo et al. presented two complementary techniques for foiling stack smashing attacks that attempt to corrupt the return address[2]. The techniques are implemented as dynamically loaded libraries Libsafe and Libverify. Libsafe replaces vulnerable C library functions with safe implementations. Libverify implements a return address verification scheme similar to StackGuard;

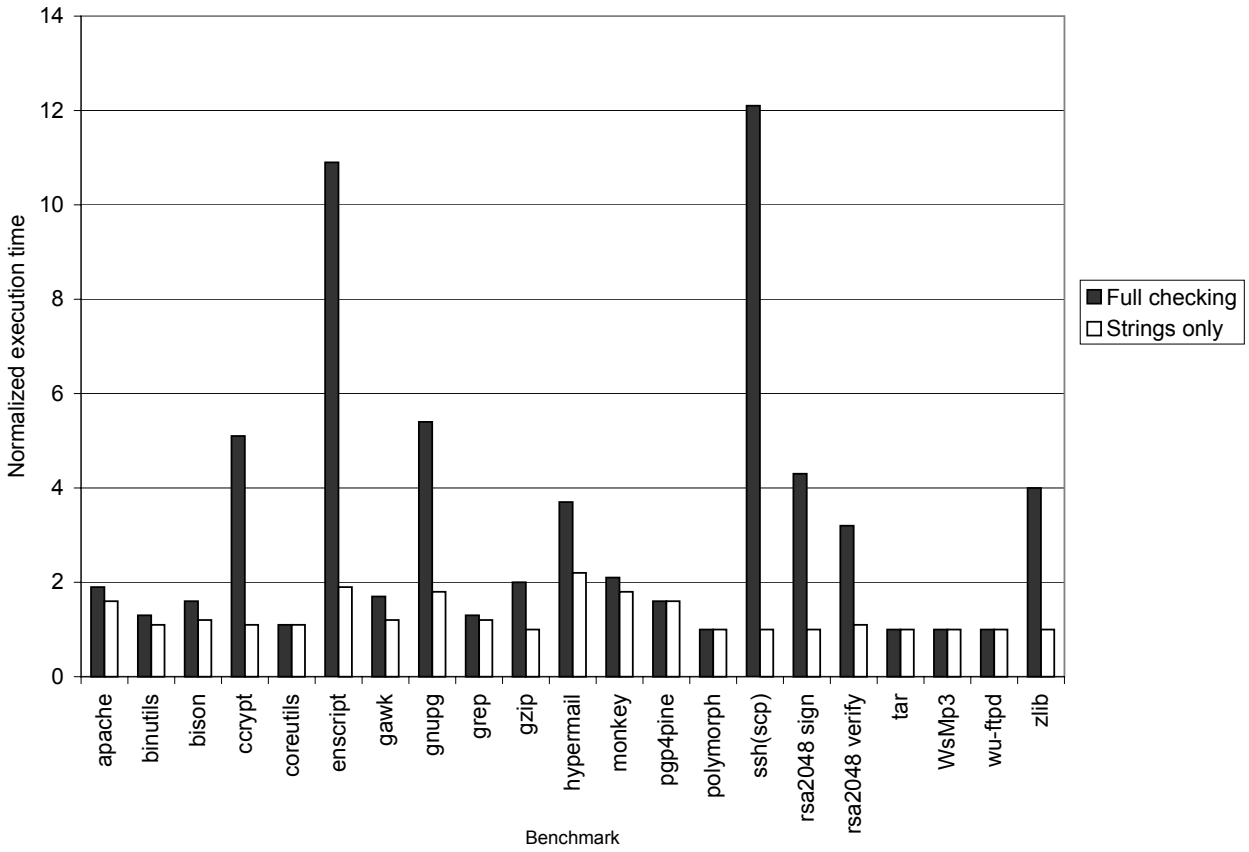

Figure 5. Performance overhead of instrumentation with and without strings only checking optimization. Non instrumented code is normalized to 1.

however it works on executables and, as such, does not require recompilation of source code, making it applicable to legacy code. A combination of both tools is ineffective against overflows that is caused by dereferencing out-of-bounds addresses, which our tool correctly detects.

Lhee and Chapin presented a buffer overflow detection technique using array bounds checking[23]. In their scheme object files are augmented with type information about static and automatic buffers that is used to carry out this range checking. Their technique does not guard against overflows caused by erroneous pointer arithmetic, making it an impractical solution. Our tool offers this protection.

Haugh and Bishop presented STOBO, an instrumentation tool that aids detection of buffer overflow vulnerabilities due to use of C library functions during testing[16]. STOBO keeps track of lengths of memory buffers, checks if they satisfy certain conditions when used as arguments to library functions and issues warnings when buffer overflows may occur from such uses. STOBO finds vulnerabilities in programs even when the test data does not trigger and overflow. However it detects only vulnerabilities due to use of library functions. It also generates false alarms.

5.3. Combination of Static and Dynamic Analysis

Necula et al. presented a program transformation tool (CCured) that adds memory safety guarantees to C programs[24, 9]. CCured first attempts to statically verify the absence of memory errors in a program by enforcing a strong type system. It then inserts run-time checks to handle portions of the code for which static verification is insufficient. CCured is incompatible with complex C code, therefore manual intervention in the form of annotations and source code changes is required for the system

to work with real-life programs. Our tool is fully automatic and compatible with complex C code.

Cyclone by Jim et al. is a safe dialect of C which prevents memory errors by using static analysis and runtime checks in a similar manner to CCured [17]. Cyclone changes pointer representation and is therefore incompatible with legacy code. Also source code changes are required to make Cyclone work with real-life programs.

Kiriansky et al. presented *program shepherding*, a technique that monitors control flow transfers during program execution in order to enforce a security policy[21, 20]. Program shepherding builds a custom security policy for the target program using automatic static and dynamic analyses. Buffer overflows attack are therefore prevented because a successful attack would require a control flow transfer that would violate the security policy. This technique was implemented in the DynamoRIO dynamic code modification system[3, 4]. The system works on unmodified native binaries and requires no special hardware or operating system support. However, it does not support self-modifying code.

6. Conclusions

We presented CRED, a practical dynamic buffer overflow detector for C programs. Our solution is built upon Jones and Kelly's technique of tracking the referent object of each pointer. Compared to the previous system, our solution does not break existing codes that compute with out-of-bound addresses and is significantly more efficient by limiting the buffer overrun checks to strings.

Our experimental results demonstrate the compatibility of our technique with commonly used programs and its effectiveness in detecting buffer overrun attacks in vulnerable programs. The overheads experienced range from 1% to 130%, depending on the use of strings in the application. These numbers are significantly better than previously published results on dynamic bounds checking. Further improvement in performance is possible using known techniques for optimizing bounds checking[15], and using static verification to reduce the portion of code that requires instrumentation[9].

7. Acknowledgements

We thank Herman ten Brugge, John Wilander and members of the open source development community for their assistance throughout this project. We also thank David Heine, David Brumley, Constantine Sapuntzakis, Andrew Myers, and the anonymous reviewers for their helpful comments on this paper.

References

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

[2] A. Baratloo, N. Singh, and T. Tsai. Transparent runtime defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 251–262, June 2000.

[3] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2001.

[4] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. *International Symposium on Code Generation and Optimization (CGO-03)*, March 2003.

[5] H. Brugge. Bounds checking C compiler http://web.inter.nl.net/hcc/haj.ten.brugge/.

[6] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*, 10(56), May 2000.

[7] CERT/CC. Advisories 2002. http://www.cert.org/advisories.

[8] CERT/CC. Vulnerability notes by metric http://www.kb.cert.org/vuls/bymetric.

[9] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, June 2003.

[10] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for vulnerability of the decade. In *Proceedings of DARPA Information Survivability Conference and Exposition*, pages 119–129, January 2000.

[11] 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*, pages 155–167, June 2003.

[12] H. Etoh and K. Yoda. Protecting from stack-smashing attacks http://www.trl.ibm.com/projects/ security/ssp/main.html.

[13] D. Evans, J. Guttag, J. Horning, and Y. Tan. Lclint: A tool for using specifications to check code. In *Proceedings of the SIGSOFT Symposium on Foundations of Software Engineering*, pages 87 – 96, December 1994.

[14] P. A. Fayolle and V. Glaume. A buffer overflow study, attacks and defenses http:// downloads.securityfocus.com/library/report.pdf.

[15] R. Gupta. Optimizing array bounds checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1-4):135–150, March–December 1993.

[16] E. Haugh and M. Bishop. Testing C programs for buffer overflow vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, February 2003.

[17] T. Jim, G. Morriset, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, pages 275 – 288, June 2002.

[18] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the International Workshop on Automatic Debugging*, pages 13–26, May 1997.

[19] S. C. Kendall. Bcc: Run-time checking for C programs. In *Proceedings of the USENIX Summer Conference*, pages 5–16, 1983.

[20] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the USENIX Security Symposium*, pages 191–206, August 2002.

[21] V. Kiriansky, D. Bruening, and S. Amarasinghe. Execution model via program shepherding, www.cag.lcs.mit.edu/commit/papers/ 03/rio-security-tm-638.pdf, May 2003.

[22] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the USENIX Security Symposium*, pages 177–190, August 2001.

[23] K. S. Lhee and S. J. Chapin. Type-assisted dynamic buffer overflow detection. In *Proceedings of the USENIX Security Symposium*, pages 81–89, August 2002.

[24] S. McPeak, G. C. Necula, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pages 128 – 139, January 2002.

[25] D. Mosberger and T. Jin. httperf - a tool for measuring web server performance http://www.hpl.hp.com/personal/david_mosberger/httperf.html.

[26] Securiteam. http://www.securiteam.com.

[27] SecurityFocus. http://online.securityfocus.com/bid.

[28] I. Simon. A comparative analysis of methods of defense against buffer overflow attacks http://www.mcs.csuhayward.edu/šimon/security/boflo.html.

[29] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

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

[31] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and Distributed Systems Security Symposium*, pages 3–7, February 2000.

[32] D. A. Wheeler. Sloccount http://www.dwheeler.com/sloccount/.

[33] J. Wilander and M. Kamkar. A comparison of publicly available tools for static intrusion detection. In *Proceedings of the Nordic Workshop on Secure IT Systems*, pages 68–84, November 2002.

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