Dynamic Buffer Overflow Detection*

Michael Zhivich
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02420
mzhivich@ll.mit.edu

Tim Leek
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02420
tleek@ll.mit.edu

Richard Lippmann
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02420
lippmann@ll.mit.edu

ABSTRACT

The capabilities of seven dynamic buffer overflow detection tools (Chaperon, Valgrind, CCured, CRED, Insure++, ProPolice and TinyCC) are evaluated in this paper. These tools employ different approaches to runtime buffer overflow detection and range from commercial products to open-source gcc-enhancements. A comprehensive testsuite was developed consisting of specifically-designed test cases and model programs containing real-world vulnerabilities. Insure++, CCured and CRED provide the highest buffer overflow detection rates, but only CRED provides an open-source, extensible and scalable solution to detecting buffer overflows. Other tools did not detect off-by-one errors, did not scale to large programs, or performed poorly on complex programs.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging; K.4.4 [**Computers and Society**]: Electronic Commerce Security

General Terms

Measurement, Performance, Security, Verification

Keywords

Security, buffer overflow, dynamic testing, evaluation, exploit, test, detection, source code

*This work was sponsored by the Advanced Research and Development Activity under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

2005 Workshop on the Evaluation of Software Defect Detection Tools 2005, June 12, Chicago, IL.

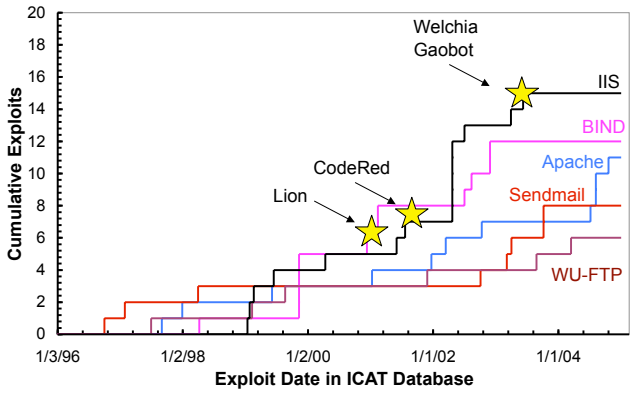

Figure 1: Cumulative exploits in commonly used server software.

1. INTRODUCTION

Today's server software is under constant scrutiny and attack, whether for fun or for profit. Figure 1 shows the cumulative number of exploits found in commonly used server software, such as `IIS`, `BIND`, `Apache`, `sendmail`, and `wu-ftpd`. The stars indicate appearances of major worms, such as `Lion`, `CodeRed` and `Welchia`. As the data demonstrates, new vulnerabilities are still found, even in code that has been used and tested for years. A recent analysis by Rescorla [18] agrees with this observation, as it shows that vulnerabilities continue to be discovered at a constant rate in many types of software.

Buffer overflows enable a large fraction of exploits targeted at today's software. Such exploits range from arbitrary code execution on the victim's computer to denial of service (DoS) attacks. For 2004, CERT lists 3,780 vulnerabilities [3], while NIST reports that 75% of vulnerabilities in its ICAT database are remotely exploitable, of which 21% are due to buffer overflows [15]. Detecting and eliminating buffer overflows would thus make existing software far more secure.

There are several different approaches for finding and preventing buffer overflows. These include enforcing secure coding practices, statically analyzing source code, halting exploits via operating system support, and detecting buffer overflows at runtime [5]. Each approach has its advantages; however, each also suffers from limitations. Code reviews, no matter how thorough, will miss bugs. Static analysis seems

like an attractive alternative, since the code is examined automatically and no test cases are required. However, current static analysis tools have unacceptably high false alarm rates and insufficient detection rates [24]. Operating system patches, such as marking stack memory non-executable, can only protect against a few types of exploits.

Dynamic buffer overflow detection and prevention is an attractive approach, because fundamentally there can be no false alarms. Tools that provide dynamic buffer overflow detection can be used for a variety of purposes, such as preventing buffer overflows at runtime, testing code for overflows, and finding the root cause of segfault behavior.

One disadvantage of using this approach to find errors in source code is that an input revealing the overflow is required, and the input space is generally very large. Therefore, dynamic buffer overflow detection makes the most sense as part of a system that can generate these revealing inputs. This evaluation is part of a project to create a grammar-based dynamic program testing system that enables buffer overflow detection in server software before deployment. Such a testing system will use the dynamic buffer overflow detection tool to find buffer overflows on a range of automatically-generated inputs. This will enable a developer to find and eliminate buffer overflows before the faults can be exploited on a production system. A similar testing approach is used in the PROTOS project at the University of Oulu [13].

This paper focuses on evaluating the effectiveness of current dynamic buffer overflow detection tools. A similar evaluation has been conducted by Wilander et al. [22], but it focused on a limited number of artificial exploits which only targeted buffers on the stack and in the bss section of the program. Our evaluation reviews a wider range of tools and approaches to dynamic buffer overflow detection and contains a more comprehensive test corpus.

The test corpus consists of two different testsuites. Section 3 presents the results for *variable-overflow* testsuite, which consists of 55 small test cases with variable amounts overflow, specifically designed to test each tool's ability to detect small and large overflows in different memory regions. Section 4 presents the results for 14 model programs containing remotely exploitable buffer overflows extracted from `bind`, `wu-ftpd` and `sendmail`.

The rest of the paper is organized as follows: Section 2 presents an overview of the tools tested in this evaluation, Sections 3 and 4 present descriptions and results for two different testsuites, Section 5 describes performance overhead incurred by the tools in this evaluation, and Section 6 summarizes and discusses our findings.

2. DYNAMIC BUFFER OVERFLOW DETECTION TOOLS

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

A summary of tool characteristics is presented in Table 1. A tool is considered to include *fine-grained bounds checking* if it can detect small (off-by-one) overflows. A tool *compiles large programs* if it can be used as a drop-in replacement for gcc and no changes to source code are needed to build the executable; however, minimal changes to the makefile are acceptable. The *time of error reporting* specifies whether the error report is generated when the error occurs or when the program terminates. Since program state is likely to become corrupted during an overflow, continuing execution after the first error may result in incorrect errors being reported. Instrumentation may also be corrupted, causing failures in error checking and reporting. If a tool can protect the program state by intercepting out-of-bounds writes before they happen and discarding them, reporting errors at termination may provide a more complete error summary.

2.1 Executable Monitoring Tools

Chaperon [16] is part of the commercial Insure toolset from Parasoft. Chaperon works directly with binary executables and thus can be used when source code is not available. It intercepts calls to `malloc` and `free` and checks heap accesses for validity. It also detects memory leaks and *read-before-write* errors. One limitation of Chaperon is that fine-grained bounds checking is provided only for heap buffers. Monitoring of buffers on the stack is very coarse. Some overflows are reported incorrectly because instrumentation can become corrupted by overflows. Like all products in the Insure toolset, it is closed-source which makes extensions difficult.

Valgrind [12] is an open-source alternative to Chaperon. It simulates code execution on a virtual x86 processor, and like Chaperon, intercepts calls to `malloc` and `free` that allow for fine-grained buffer overflow detection on the heap. After the program in simulation crashes, the error is reported and the simulator exits gracefully. Like Chaperon, Valgrind suffers from coarse stack monitoring. Also, testing is very slow (25 – 50 times slower than running the executable compiled with gcc [12]), since the execution is simulated on a virtual processor.

2.2 Compiler-based Tools

CCured [14] works by performing static analysis to determine the type of each pointer (`SAFE`, `SEQ`, or `WILD`). `SAFE` pointers can be dereferenced, but are not subject to pointer arithmetic or type casts. `SEQ` pointers can be used in pointer arithmetic, but cannot be cast to other pointer types, while `WILD` pointers can be used in a cast. Each pointer is instrumented to carry appropriate metadata at runtime - `SEQ` pointers include upper and lower bounds of the array they reference, and `WILD` pointers carry type tags. Appropriate checks are inserted into the executable based on pointer type. `SAFE` pointers are cheapest since they require only a `NULL` check, while `WILD` pointers are the most expensive, since they require type verification at runtime.

The main disadvantage of CCured is that the programmer may be required to annotate the code to help CCured determine pointer types in complex programs. Since CCured requires pointers to carry metadata, wrappers are needed to strip metadata from pointers when they pass to uninstrumented code and create metadata when pointers are received from uninstrumented code. While wrappers for commonly-used C library functions are provided with CCured, the developer will have to create wrappers to interoperate with other uninstrumented code. These wrappers introduce another source of mistakes, as wrappers for `sscanf` and `fscanf` were incorrect in the version of CCured tested in this evaluation; however, they appear to be fixed in the currently-available version (v1.3.2).

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

Table 1: Summary of Tool Characteristics (* = fine-grained bounds checking on heap only)

C Range Error Detector (CRED) [19] has been developed by Ruwase and Lam, and builds on the Jones and Kelly "referent object" approach [11]. An *object tree*, containing the memory range occupied by all objects (i.e. arrays, structs and unions) in the program, is maintained during execution. When an object is created, it is added to the tree and when it is destroyed or goes out of scope, it is removed from the tree. All operations that involve pointers first locate the "referent object" – an object in the tree to which the pointer currently refers. A pointer operation is considered illegal if it results in a pointer or references memory outside said "referent object." CRED's major improvement is adhering to a more relaxed definition of the C standard – *out-of-bounds* pointers are allowed in pointer arithmetic. That is, an *out-of-bounds* pointer can be used in a comparison or to calculate and access an *in-bounds* address. This addition fixes false alarms that were generated in several programs compiled with Jones and Kelly's compiler, as pointers are frequently tested against an *out-of-bounds* pointer to determine a termination condition. CRED does not change the representation of pointers, and thus instrumented code can interoperate with unchecked code.

Two main limitations of CRED are unchecked accesses within library functions and treatment of structs and arrays as single memory blocks. The former issue is partially mitigated through wrappers of C library functions. The latter is a fundamental issue with the C standard, as casting from a struct pointer to a char pointer is allowed. When type information is readily available at compile time (i.e. the buffer enclosed in a struct is accessed via `s.buffer[i]` or `s_ptr->buffer[i]`), CRED detects overflows that overwrite other members within the struct. However, when the buffer inside a struct is accessed via an alias or through a type cast, the overflow remains undetected until the boundary of the structure is reached. These problems are common to all compiler-based tools, and are described further in Section 2.3.

Insure++ [16] is a commercial product from Parasoft and is closed-source, so we do not know about its internal workings. Insure++ examines source code and inserts instrumentation to check for memory corruption, memory leaks, memory allocation errors and pointer errors, among other things. The resulting code is executed, and errors are reported when they occur. Insure's major fault is its performance overhead, resulting in slowdown factor of up to 250 as compared to gcc. Like all tools, Insure's other limitation stems from the C standard, as it treats structs and arrays as single memory blocks. Since the product is closed-source, extensions are difficult.

ProPolice [8] is similar to StackGuard [6], and outperformed it on artificial exploits [22]. It works by inserting a "canary" value between the local variables and the stack frame whenever a function is called. It also inserts appropriate code to check that the "canary" is unaltered upon return from this function. The "canary" value is picked randomly at compile time, and extra care is taken to reorder local variables such that pointers are stored lower in memory than stack buffers.

The "canary" approach provides protection against the classic "stack smashing attack" [1]. It does not protect against overflows on the stack that consist of a single out-of-bounds write at some offset from the buffer, or against overflows on the heap. Since ProPolice only notices the error when the "canary" has changed, it does not detect read overflows or underflows. The version of ProPolice tested during this evaluation protected only functions that contained a character buffer, thus leaving overflows in buffers of other types undetected; this problem has been fixed in later versions by including `-fstack-protector-all` flag that forces a "canary" to be inserted for each function call.

Tiny C compiler (TinyCC) [2] is a small and fast C compiler developed by Fabrice Bellard. TinyCC works by inserting code to check buffer accesses at compile time; however, the representation of pointers is unchanged, so code compiled with TinyCC can interoperate with unchecked code compiled with gcc. Like CRED, TinyCC utilizes the "referent object" approach [11], but without CRED's improvements. While TinyCC provides fine-grained bounds checking of buffer accesses, it is much more limited than gcc in its capabilities. It failed to compile large programs such as `Apache` with the default makefile. It also does not detect read overflows, and terminates with a segfault whenever an overflow is encountered, without providing an error report.

2.3 Common Limitations of Compiler-based Tools

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

The latter problem is a fundamental issue with the C stan-

dard of addressing memory in arrays and structs. According to the C standard, a pointer to any object type can be cast to a pointer to any other object type. The result is defined by implementation, unless the original pointer is suitably aligned to use as a resultant pointer [17]. This allows the program to re-interpret the boundaries between struct members or array elements; thus, the only way to handle the situation correctly is to treat structs and arrays as single memory objects. Unfortunately, overflowing a buffer inside a struct can be exploited in a variety of attacks, as the same struct may contain a number of exploitable targets, such as a function pointer, a pointer to a `longjmp` buffer or a flag that controls some aspect of program flow.

3. VARIABLE-OVERFLOW TESTSUITE EVALUATION

The *variable-overflow* testsuite evaluation is the first of two evaluations included in this paper. This testsuite is a collection of 55 small C programs that contain buffer overflows and underflows, adapted from Misha Zitser's evaluation of static analysis tools [24]. Each test case contains either a *discrete* or a *continuous* overflow. A *discrete* buffer overflow is defined as an out-of-bounds write that results from a single buffer access, which may affect up to 8 bytes of memory, depending on buffer type. A *continuous* buffer overflow is defined as an overflow resulting from multiple consecutive writes, one or more of which is out-of-bounds. Such an overflow may affect an arbitrary amount of memory (up to 4096 bytes in this testsuite), depending on buffer type and length of overflow.

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

3.1 Test Procedure

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

Table 1 describes the versions of tools tested in our evaluation. All tests were performed on a Red Hat Linux release 9 (Shrike) system with dual 2.66GHz Xeon CPUs. The standard Red Hat Linux kernel was modified to ensure that the location of the stack with respect to *stacktop* address (`0xC0000000`) remained unchanged between executions. This modification was necessary to ensure consistent segfault behavior due to large overflows.

3.2 Variable-overflow Testsuite Results

This section presents a summary of the results obtained with the variable-overflow testsuite. The graph in Figure 2 shows the fraction of test cases in the variable-overflow testsuite with a non-*miss* (*detection* or *segfault*) outcome for each amount of overflow. Higher fractions represents better performance. All test cases, with the exception of the 4 underflow test cases, are included on this graph even though the proportional composition of the testsuite is not representative of existing exploits. Nonetheless, the graph gives a good indication of tool performance. Fine-grained bounds checking tools are highlighted by the "fine-grained" box at the top of the graph.

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

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

4. REAL EXPLOIT EVALUATION

Previously, we evaluated the ability of a variety of tools employing *static analysis* to detect buffer overflows [25]. These tools ranged from simple lexical analyzers to abstract interpreters [9, 10, 20, 21, 23]. We chose to test these tools against fourteen historic vulnerabilities in the popular Internet servers `bind`, `sendmail`, and `wu-ftpd`. Many of the detectors were unable to process the entire source for these programs. We thus created *models* of a few hundred lines that reproduced most of the complexity present in the original. Further, for each model, we created a patched copy in which we verified that the overflow did not exist for a test input that triggered the error in the unpatched version. In that evaluation, we found that current static analysis tools

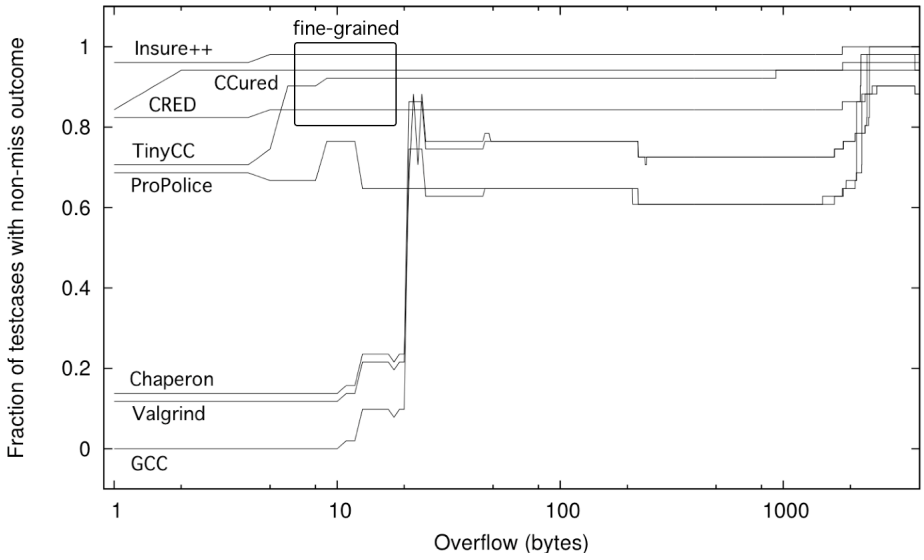

Figure 2: Combined fraction of detections and segfaults vs the amount of overflow in bytes. A box highlights tools with fine-grained bounds checking capabilities.

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

4.1 Test Procedure

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

4.2 Real Exploit Results

Table 2 presents the results of this evaluation, which agree well with those on the variable-overflow testsuite. Three of the dynamic overflow detectors that provide fine-grained bounds checking, CCured, CRED, and TinyCC, work extremely well, detecting about 90% of the overflows whilst raising only one false alarm each. The commercial program Insure, which also checks bounds violations rigorously, fares somewhat worse with both fewer detections and more false alarms. Notice that misses and false alarms for these tools are errors in the implementation, and are in no way a fundamental limitation of dynamic approaches. For example, in the case of CRED the misses are due to an incorrect `memcpy` wrapper; there are no misses once this wrapper is corrected. The CRED false alarm is the result of overly aggressive string length checks included in the wrappers for string manipulation functions such as `strchr`. None of the tools are given credit for a segmentation fault as a signal of buffer overflow (except TinyCC and gcc as this is the only signal provided). This is why, for instance, ProPolice appears to perform *worse* than gcc. As a final comment, it is worth considering the performance of gcc alone. If provided with the right input, the program itself detects almost half of these real overflows, indicating that input generation may be a fruitful area of future research.

5. PERFORMANCE OVERHEAD

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

Our evaluation tests overhead on two common utility programs (`gzip` and `tar`), an encryption library (`OpenSSL`) and a webserver (`Apache`). For `OpenSSL` and `tar`, the testsuites included in the distribution were used. The test for `gzip` consisted of compressing a tar archive of the source code package for glibc (around 17MB in size). The test for `Apache` consisted of downloading a 6MB file 1,000 times on a loopback connection. The overhead was determined by timing the execution using `time` and comparing it to executing the test when the program is compiled with gcc. The results are summarized in Table 3. Programs compiled with gcc executed the tests in 7.2s (`gzip`), 5.0s (`tar`), 16.9s (`OpenSSL`) and 38.8s (`Apache`).

Compiling and running `Apache` presented the most problems. Chaperon requires a separate license for multi-threaded programs, so we were unable to evaluate its overhead. Valgrind claims to support multi-threaded programs but failed to run due to a missing library. Insure++ failed on the configuration step of the makefile and thus was unable to

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

Table 2: Dynamic buffer overflow detection in 14 models of real vulnerabilities in open source server code. There are four `bind` models (b1–b4), three `wu-ftpd` models (f1–f3), and seven `sendmail` models (s1–s7). A 'd' indicates a tool detected a historic overflow, while an 'f' means the tool generated a false alarm on the patched version. $P(det)$ and $P(fa)$ are the fraction of model programs for which a tool signals a detection or false alarm, respectively.

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

Table 3: Instrumentation overhead for commonly used programs as a multiple of gcc execution time. The blank entries indicate that the program could not be compiled or executed with the corresponding tool.

compile `Apache`. CCured likewise failed at configuration, while TinyCC failed in parsing one of the source files during the compilation step.

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

While CRED incurs large overhead on programs that involve many buffer manipulations, it has the smallest overhead for a fine-grained bounds checking tool. CRED can be used as a drop-in replacement for gcc, as it requires no changes to the source code in order to compile these programs. Only minimal changes to the makefile were required to enable bounds-checking and turn off optimizations. CRED's high detection rate, ease of use and relatively small overhead make it the best candidate for use in a comprehensive solution for dynamic buffer overflow detection.

6. DISCUSSION

The three top-performing tools in our evaluation are Insure++, CCured and CRED. Insure++ performs well on test cases, but not on model programs. It adds a large performance overhead, and the closed-source nature of the tool inhibits extensions. CCured shows a high detection rate and is open-source; however, it requires rewriting 1–2% of code to compile complicated programs [4]. CRED also offers a high detection rate, and it is open-source, easily extensible and has fairly low performance overhead (10% slowdown for simple Apache loopback test). Its main disadvantage is lack of overflow detection in library functions compiled without bounds-checking. Like all compiler-based tools, CRED does not detect overflows within structs in a general case; however, if the buffer enclosed in a struct is referenced directly, then CRED detects the overflow.

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

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

7. REFERENCES

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

[2] F. Bellard. TCC: Tiny C compiler. http://www.tinycc.org, Oct. 2003.

[3] CERT. CERT/CC statistics. http://www.cert.org/stats/cert_stats.html, Feb. 2005.

[4] 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*, pages 232–244. ACM Press, 2003.

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

[6] 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*, pages 63–78, San Antonio, Texas, Jan. 1998.

[7] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, Dec. 2004.

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

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

[10] G. Holzmann. Static source code checking for user-defined properties. In *Proc. IDPT 2002*, Pasadena, CA, USA, June 2002.

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

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

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

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

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

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

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

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

[19] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *Proceedings of Network and Distributed System Security Symposium*, pages 159–169, 2004.

[20] P. Technologies. PolySpace C verifier. http://www.polyspace.com/c.htm, Sept. 2001.

[21] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, Feb. 2000.

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

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

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

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