

Defending against Buffer-Overflow Vulnerabilities

Bindu Madhavi Padmanabhuni and Hee Beng Kuan Tan
Nanyang Technological University, Singapore

A survey of techniques ranging from static analysis to hardware modification describes how various defensive approaches protect against buffer overflow, a vulnerability that represents a severe security threat.

In 2003, an analysis of buffer overflow pronounced it the vulnerability of the decade.¹ The following year, the Open Web Application Security Project (OWASP) listed it as the fifth most serious Web application weakness. In the first five months of 2010, the National Vulnerability Database (<http://nvd.nist.gov>) recorded 176 buffer overflow vulnerabilities, of which 136 had a high severity rating. Buffer overflow remains a major security hole today, ranking third on the Common Weakness Enumeration/SANS list of Top 25 Most Dangerous Software Errors (<http://cwe.mitre.org/top25>).

Buffer overflow occurs during program execution when an application writes beyond the bounds of a preallocated fixed-size buffer. This data overwrites adjacent memory locations and, depending on what it overwrites, can affect program behavior. The lack of bounds-checking operations for filling the buffers permits this error. Applications written in programming languages such as C or C++ are commonly associated with buffer-overflow vulnerabilities because they allow overwriting any part of memory without checking whether the data written will overflow its allocated memory.

A review of buffer-overflow exploits and an analysis of their solutions reveals deficiencies in present defenses, providing a basis for developing modifications to protect against such exploits.

BUFFER-OVERFLOW EXPLOITS

Attackers can use buffer overflows to launch denial-of-service (DoS) attacks, spawn a root shell, gain higher-order access rights (especially root or administrator privileges), steal information, or impersonate a user. In 1998, the Morris worm, one of the first to strike the Internet, exploited a buffer overflow in the Unix finger daemon (fingerd) to propagate itself from one machine to another (http://en.wikipedia.org/wiki/Morris_worm). The 2001 Code Red worm took advantage of the same weakness in the Microsoft IIS webserver and reportedly infected 359,000 systems within 14 hours; (see [http://en.wikipedia.org/wiki/Code_Red_\(computer_worm\)](http://en.wikipedia.org/wiki/Code_Red_(computer_worm))). In 2003, SQL Slammer exploited a buffer overflow in the Microsoft SQL server, spread quickly, and launched a DoS attack on various targeted networks (http://en.wikipedia.org/wiki/SQL_slammer).

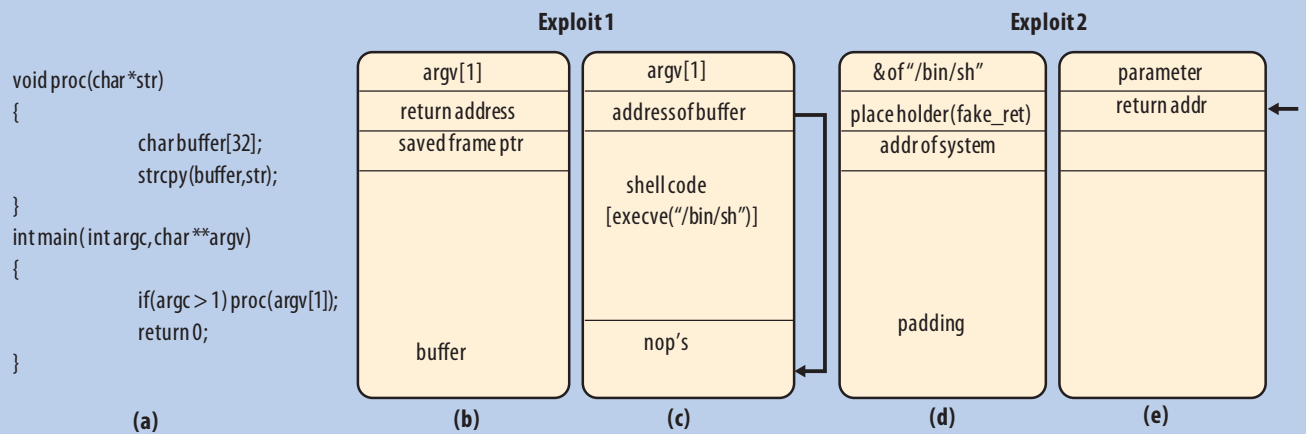


Figure 1. Examples of function activation record exploits. (a) Vulnerable code snippet. If `str` is longer than 31, it modifies the memory area next to it. (b) Stack frame of `proc` before `strcpy(buffer, str)`. (c) Stack frame after `strcpy(buffer, str)`. Exploit 1 uses a string consisting of shell code and the memory address to which the attack code copies the shell code to cause a buffer overflow. When the function returns, it jumps to the shell code to spawn a shell. Exploit 2 is a return-to-libc attack on the code snippet that spawns a shell by overwriting the return address with the address of `system()`. (d) Stack frame after buffer overflow and before program returns to `system()`. The attack code includes the address for `system()` as well as its parameters. (e) Stack frame after program returns to `system()`.

To carry out an exploit, attackers must find suitable code to attack and make program control jump to that location with the required data in memory and registers. Attackers glean information about the vulnerable program code and its runtime behavior from the program's documentation and source code, by disassembling a binary file, or by running the program in a debugger. Buffer overflow exploits generally target function activation records, pointers, or the management data of heap-based memory blocks.

Function activation record exploits

A popular technique targets the function activation record. When program execution calls a function, stack frame is allocated with function arguments, return address, the previous frame pointer, saved registers, and local variables. In the stack frame, the return address points to the next instruction for execution after the current function returns. Attackers can overflow a buffer on the stack beyond its allocated memory and modify the return address to change program control to a location of their choice. Figure 1 shows two examples of these attacks.

Instead of supplying executable code, an attacker can supply data to a C library function, such as `system()`, that is already present in the program code. Such exploits are called return-to-libc attacks because they direct control to a C library function rather than to attacker-injected code; they also alter the return address. Return-to-libc attacks are ideal for exploiting programs that have memory protection mechanisms, like nonexecutable stacks, because they do not execute attacker-supplied code.

Another target is the previous frame pointer. An attacker can build a fake stack frame with a return address pointing

to the attacker's code. An overflow of the previous frame pointer will point to this fake stack frame. When the function returns, the attacker's code executes.

Pointer subterfuge exploits

Pointer subterfuge exploits involve modifying pointer values, such as function, data, or virtual pointers; they also can modify exception handlers. Consider the following code snippet, which includes a buffer that has a function pointer allocated in the data section:

```
char buf[64];
int (*pfn)(char*) = NULL;
void main(int argc, char **argv)
{
    ...
    strcpy(buf, argv[1]);
    iResult = (*pfn)(argv[2]);
    ...
}
```

An attack can use `strcpy()` to overflow the buffer and overwrite the function pointer. The overwritten pointer can point to the address of shell code or `system()`. The attack takes place when the program calls the function pointer.

Attackers can use pointer subterfuge in overruns of stacks, heaps, or objects containing embedded function pointers. This kind of attack is especially effective when the program uses methods for preventing return address modification because it does not change the saved return address.

An exploit can use data pointers to indirectly modify the return address. Such indirect overwriting schemes are

Example exploit

```
classVulnerable : public SomeBase
{
public:
    charbuffer[100];
    virtual void Func1();
    virtual void Func2();
}
void main()
{
    ...
    Vulnerable v;
    std::cin>>v.buffer;
    v.Func1();
    ...
}
```

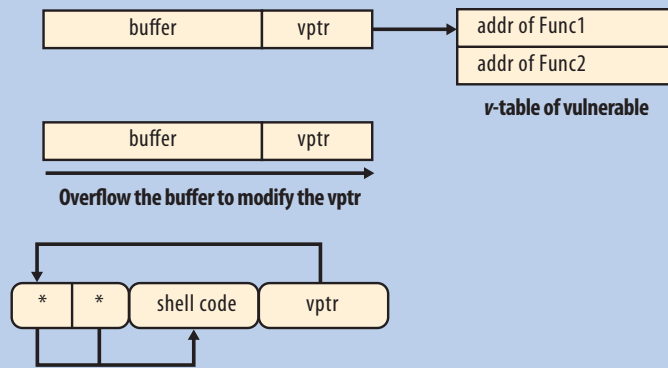


Figure 2. Sample code vulnerable to virtual-function pointer smashing. An attack can overflow the object’s member variable `buffer` to modify the `vptr`, making it point to an attacker-supplied virtual table with injected code. Control will then transfer to this code with the next call to the virtual function.

useful if the program uses a protection mechanism like StackGuard¹ because they alter the return address without changing the canary—a value placed in the stack. When a buffer in the stack overflows, it will corrupt the canary. A program can use the canary as a check against buffer overflow.

Another method for hijacking program control uses `longjmp` buffers. The C standard library provides `setjmp/longjmp` to perform nonlocal jumps. Function `setjmp` saves the calling function’s environment into the `jmp_buf` type variable (which is an array type) for later use by `longjmp`, which restores the environment from the most recent invocation of the `setjmp` call. An attacker can overflow the `jmp_buf` with the address of the attacker’s code; when the program calls `longjmp`, it will jump to the attacker’s code.

Although not widely used, virtual-function pointer smashing is a threat even when a program uses antistack-smashing protection, because such protection does not defend against overflow in the heap. C++ compilers use tables to implement virtual functions. These tables have an array of function pointers that the program uses at runtime to implement dynamic binding. Each instantiated object has a virtual pointer pointing to its virtual table as part of an object’s header. By making the virtual pointer point to an attacker-supplied virtual table with injected code, an attacker can transfer control to this code at the virtual function’s next call.

Figure 2 shows sample code vulnerable to virtual-function pointer smashing. This attack overflows the object’s member variable `buffer` to modify the `vptr`, making it point to an attacker-supplied virtual table with injected

code. Control will then transfer to this code with the next call to the virtual function.

The Microsoft Windows Structured Exception Handling (SEH) mechanism is also an exploit target. When the program generates an exception, Windows SEH will catch it if the program has no handler or if the provided handler cannot process the exception. The function pointer for the exception handler is on the stack. By overflowing the stack buffer, an attacker can modify it to transfer control to another location.

Heap-based exploits

Dynamic memory allocators such as `malloc` allocate memory on the heap dynamically during runtime. Linked lists manage memory blocks that are allocated and deallocated dynamically using `malloc` and `free`. The management data for each memory block, such as its size and pointers to other memory chunks, is stored in a linked-list-like data structure. The user data and management data are adjacent in a chunk similar to local variables and the return address on a stack. By overflowing user data in the memory block, an attack can corrupt the management data. However, modifying such data does not change program control because this data is not a pointer. Heap-based exploits corrupt the metadata of heap-allocated memory blocks and use it to change other pointers.

DEFENSIVE TECHNIQUES

Researchers have proposed various approaches to address buffer overflow problems, ranging from best practices in development to automated frameworks for recovering from attacks. The five basic methods include

defensive coding practices, runtime instrumentation, static code analysis, combined static and dynamic code analysis, and network-based instrumentation.

Defensive coding practices

Writing secure code by applying suitable defensive coding practices is the best solution for eliminating vulnerabilities. C and C++ provide no built-in protection for detecting out-of-bound memory accesses. Choosing programming languages like Java or environments like .NET that perform runtime bounds checking eliminates the problem. Standard C library functions including `strcpy`, `strcat`, and `gets` are unsafe because they do not perform bounds checking. Using safer versions of these functions such as `strcpy_s` and `strcat_s` is another good defensive coding practice.

Although defensive coding practices are helpful in developing more secure programs, they are not always feasible in practice.

Performing bounds checking on all arrays solves the problem but decreases performance. Programmers must employ heuristics to identify security-critical buffers and then apply bounds checking to those buffers. Although defensive coding practices are helpful in developing more secure programs, they are not always feasible in practice and are applicable only when actually writing (or planning to write) code. Defensive coding will not work when the source code is unavailable or must remain the same.

Runtime instrumentation

Many runtime techniques for defending against attacks use return address modification to detect buffer overflows. Some proposals include obtaining information about buffer bounds estimates and instrumenting the code for runtime bounds checking.

Compile-time techniques like StackGuard¹ and Return Address Defender (RAD)² insert code to check for return address modification. StackGuard places a canary before the return address and checks the canary's value when the function returns. RAD creates a Return Address Repository global array and copies the return address to it in the function prologue. It then checks for modifications in the function epilogue. These approaches are not completely foolproof because attackers can alter the return address indirectly by using a pointer. StackShield stores the return address in the function prologue and transfers program control to the saved return address on function return.²

This requires source code recompilation and only defends against return-address-based attacks.

Libverify copies functions to heap memory and executes the functions from copied versions.² It uses wrapper functions to store the return address on function entry and verifies it on function return. Libverify uses the return address itself as a canary, reducing the binary instrumentation procedure because the offsets remain the same, unlike StackGuard. Libverify does not need source code but leaves the canary stack itself unprotected.

An alternative approach to Libverify's load-time code instrumentation is to insert instrument code in the executable code itself.³ Because it adds instrumented code before entry and after every function exit address, this approach requires modification of all memory references in the binary. Although it does not require source code, it cannot protect against attacks that target data structures other than the return address.

Split Stack and Secure Return Address (SAS) use two separate stacks, one for data and the other for information control.⁴ However, they do not detect buffer overflow, which corrupt a buffer's neighboring locations. SmashGuard proposes hardware modifications using modified microcoded instructions for CALL and RET opcodes.⁵ These modified instructions store and compare the return address. These two approaches do not require source code recompilation, but they only prevent exploits of frame activation records.

Hardware-supported instruction-level runtime taint analysis addresses noncontrol data attacks.⁶ This approach does not need any changes to the memory system or the processor pipeline dealing with program data. However, it requires classifying instructions as tainted and taintless.

Solar Designer² and Pax⁵ use a nonexecutable stack to combat buffer overflow. However, nonexecutable-stack methods cannot defend against return-to-libc attacks and attacks on data segments; some instances also need an executable stack. OpenWall maps the shared libraries' address space so that their addresses always contain zero bytes to defend against return-to-libc attacks. Pax uses address space layout randomization (ASLR) and a page-based mechanism to protect the heap and stack.

ProPolice places a canary before the return address and also places pointers before the local buffers.² Doing so prevents exploitation of activation records but cannot prevent heap- and data-segment-based attacks involving the function pointer, long-jump buffer variables, and members of structures because ProPolice cannot rearrange pointer variables of structures. PointGuard uses encryption to protect against code and data pointer attacks, but the encryption and decryption might significantly decrease runtime performance.¹

Libsafe intercepts all calls to unsafe standard C library functions.² It substitutes similar functions that limit any overflows within the current stack frame. Because a buffer

cannot extend beyond a stack frame, these overflow-limiting functions prevent overwriting of the return address. This method only protects those C library functions for which it has substitutes; even for these cases, an attacker can overwrite anything up to the frame pointer. Thus, this method cannot protect against attacks targeting function pointers and heap-based overflows. Some proposals extend Libsafe to intercepting calls to malloc, thereby preventing heap-based exploits as well.

Some solutions transform static buffers to dynamically allocated heap-based buffers. Any overflow to these buffers leads to a segmentation fault, which flags the attempted exploit.⁷ When such accesses occur, instead of letting the overflow occur and corrupt the memory, their compiler stores the out-of-bounds write value in a hash table. Whenever program execution references this value, the hash table information provides this stored value based on the read address, which allows the program to continue executing instead of crashing or halting. However, this approach can incur significant performance overhead and might be unsuitable for many applications.

Dytan is a taint-analysis framework in which the user specifies the taint sources, sinks, and taint level.⁶ It accesses the user-supplied library, control flow graph (CFG), and post-dominator-tree information to identify the sources and sinks, and applies the taint level to operands. It then uses the Pin tool (www.pintool.org) to produce an instrumented executable. To detect buffer overflow, users can specify the source as data read from the network and the sink as program control instructions like jump, ret, and so on. The taint markings associated with each byte or memory range, the storing of CFGs, PDOM trees for the program, and the program's related libraries result in high space and time overhead.

When these approaches detect attacks, they halt program operation—in effect, resulting in DoS—yet provide no mechanism for self-healing. Dira, a tool that can repair itself from a detected attack, maintains a log that records memory updates to track data dependencies.⁸ When an attack occurs, it uses this log to restore the program's memory to its preattack state. Dira incurs high runtime overhead because it must log each memory update and track each data dependency.

Exterminator is a runtime system for detecting, isolating, and correcting heap-based memory errors.⁹ Each object has metadata, which Exterminator uses for error isolation and detection before memory allocation. Based on the information from the error isolation algorithm, Exterminator generates a runtime patch for each error. For a buffer overflow, it pads the buffer with the maximum value encountered for this error. This approach does not need source code, and it is useful for testing or automatically correcting a deployed system. However, isolating and detecting the heap-based errors requires additional runs and increased memory consumption.

Using more secure versions of C like Cyclone helps prevent buffer-overflow attacks but would be practical only for yet-to-be-developed projects—porting legacy code to Cyclone would necessitate prohibitively costly code transformation or modification.² CCured translates C programs into CCured and establishes all pointers as either safe, sequenced, or dynamic through a constraint-based type-inference algorithm.² It uses runtime checks when static analysis is not enough to determine safety. CCured requires source code changes, but fewer than Cyclone.

Using more secure versions of C like Cyclone helps prevent buffer-overflow attacks but would be practical only for yet-to-be-developed projects.

Static code analysis

Static analysis of the program source code or disassembled binary code can identify buffer-overflow vulnerabilities. Although these techniques do not incur runtime overhead, they generate many false positives because they lack runtime information.

LCLint is an annotation-assisted static analysis tool¹⁰ that programmers can use to set preconditions and postconditions for state functions. Constraints used to describe buffer ranges include minSet, maxSet, minRead, and maxRead. When the program calls an annotated function, it checks pre- and postconditions to ensure safe access to buffers using these buffer range constraints. LCLint requires programmers to provide annotations and protects such annotated functions. Buffer integer range analysis protects only those library functions with annotations.

Vinod Ganapathy and colleagues model pointers to character buffers by four constraint variables to denote the maximum and minimum number of bytes the buffer allocates and uses.¹¹ They model integer variables using the variable's maximum and minimum values. This technique detects buffer overruns using solver and taint analysis when the maximum used value is greater than the allocated minimum or allocated maximum value for the buffer. However, it generates many false positives because of the flow-insensitive nature of the analysis.

A method that uses maximum length and used-length attributes models statements as constraints and functions as integer transfer functions.¹² Doing so converts the buffer-overflow problem to an error-checking problem by asserting the constraints and finding the reaching paths to this constraint error. However, this tool cannot perform function pointer analysis, nor can it handle arrays of pointers or user-defined structure arrays.

Marple identifies infeasible paths, examines buffers, and classifies paths that lead to buffer access as safe, vulnerable, overflow-input independent, or unknown.¹³ It does so by raising queries and propagating them backward along the control flow. Marple updates control flow at nodes where it can collect information and whenever it encounters a potential buffer overflow statement. The query terminates when it reaches program entry or an infeasible segment, or when information gathered during propagation resolves the query. Marple takes application source code as input and returns vulnerable path segments to the user, who can then develop patches. However, it uses a conservative analysis that might generate many false positives.

A method that traverses feasible program execution paths and uses the extracted information to perform context-sensitive taint analysis can detect vulnerabilities in x86 executables.¹⁴ The analysis identifies unsecure functions and classifies them as tainted sources or sensitive sinks. Taint analysis checks whether these functions pass tainted data from sources to sensitive sinks, and, if so, raises an alert. Because loops do not execute as many times as in concrete execution, this technique misses some feasible paths, causing false negatives.

Both source and binary code analysis tools and network tools should be part of a programmer's arsenal for protecting against buffer-overflow attacks.

Combined static and dynamic code analysis

Other solutions use both static and dynamic analysis to detect buffer-overflow vulnerabilities.

Researchers have proposed algorithms for selecting susceptible buffers, creating buffer overruns, and, based on the result, analyzing the application for susceptibility.¹⁵ This technique identifies locations that call unsafe library functions on local buffers and nonlibrary functions that read or copy user input. It then calculates the return address that attackers would overwrite to insert an attack string. This approach targets only exploits of the return address and cannot assess an application's susceptibility to other exploits.

Loop-extended symbolic execution (LESE) introduces new variables for representing trip counts for each loop and links them to variables representing program input.¹⁶ It combines these symbolic constraints with conditions for security policy violation and uses the results for vulnerability checking. LESE identifies buffer overflows in real-world programs by sending an initial benign input and uses that execution trace with grammar to discover vulnerabilities. Although this approach is suitable for discovering vulner-

abilities based on security predicate violation and input processing using loops, it might not be applicable for other purposes.

Network-based instrumentation

Network-based instrumentation techniques compare network data with vulnerability signatures from previous attacks, use dynamic taint analysis on network data, or inspect payloads for shell code.

TaintCheck identifies user input data from the network and performs runtime binary rewriting to track the propagation of tainted data.¹⁷ If the program uses tainted data as a jump target or as an argument for a system call, TaintCheck identifies an attack. It then generates an exploit signature by applying backward slicing to the tainted data propagation in the memory. TaintCheck also identifies the parts of the payload used in these attacks by monitoring how the vulnerable program uses each byte of payload at the processor instruction level. It can use this information to generate an attack signature or for hints to use in pattern extraction techniques. However, TaintCheck suffers from slow performance because it runs in Valgrind's emulation environment.

The Pasan prototype instruments source code to record information about the size of static and dynamically allocated buffers and to produce a memory update log.¹⁸ It uses RAD to detect buffer-overflow vulnerabilities. After detecting an attack, Pasan uses runtime information and the memory update log to perform a dependency analysis on the corrupted target address and identify the vulnerable code. Based on the type of code, Pasan either uses a safe library function or instruments the vulnerable code with bounds-checking code to generate a patch. However, the logging and bounds checking incurs a throughput penalty of 10 to 23 percent.

Vigilante traces network data dynamically by tracking the dataflow and generating a security trap when the program uses data unsafely, such as when it loads the data into the program counter or passes it as an argument to security-critical functions.¹⁹ However, Vigilante cannot detect attacks that overwrite security-sensitive information with values indirectly controlled by a worm.

SigFree checks for code in the request packet.²⁰ The idea is that buffer-overflow attacks need executable code to launch an exploit, but client requests to a server do not contain executables. Because SigFree is not based on vulnerability signature comparison, it can detect and block new attacks. On the other hand, it is not effective against DoS and return-into-libc attacks.

DETECTION TOOLS

Both source and binary code analysis tools and network tools should be part of a programmer's arsenal for protecting against buffer-overflow attacks.

Source code analysis tools

ITS4 (www.cigital.com/its4) scans C/C++ source code to identify dangerous standard library functions and uses a handler to perform risk evaluation based on the initial stored information such as checking for race conditions and parameters of unsafe string functions. Instead of parsing source code from a single build, it scans several files to look for vulnerabilities in multiple builds, thereby reducing false negatives. Programmers can use ITS4 in an integrated development environment to highlight errors within that editor. The tool is rudimentary, but it is better than the grep tool. However, it generates many false positives. Because ITS4 relies on a database of vulnerable functions, calling a vulnerable function not present in the database leads to false negatives.

The Rough Auditing Tool for Security (RATS; www.fortify.com/security-resources/rats.jsp) and Flawfinder (www.dwheeler.com/flawfinder) also employ a database to identify and flag security vulnerabilities. Both generate many false positives and false negatives because they perform only a rough analysis.

The Buffer Overrun Detection (www.cs.berkeley.edu/~daw/boon) tool converts the buffer-overflow detection problem to an integer constraint problem by modeling strings (based on their size and usage) and library functions. BOON identifies which buffer has been overrun, but because it is flow insensitive, it does not always reliably identify which statement has the fault or the path that leads to the fault.

Modelchecking Programs for Security (www.cs.berkeley.edu/~daw/mops) finds vulnerabilities by detecting violations of temporal safety properties. MOPS builds models of the program and of the security property, then identifies whether the program model satisfies the security property. It can detect buffer overflows and user privilege issues, but requires building rules to express temporal safety properties. MOPS also requires users to specify the properties to check.

Binary code analysis tools

Binary analysis tools cannot by themselves identify vulnerabilities. Rather, they evaluate performance and gather statistics about programs, thereby greatly aiding in the reverse engineering of binaries for vulnerability and malware detection.


Valgrind (<http://valgrind.org>) is a Linux-based instrumentation framework for building dynamic analysis tools. Programmers can use Valgrind to disassemble code into an intermediate representation, instrument it with analysis code, and convert the instrumented code back into binary code. Valgrind is useful for memory leak detection, memory debugging, program profiling, and thread error detection. It is suitable for both source and binary code analysis, but incurs performance penalties because of the code transformations.

Pin is a dynamic binary instrumentation tool that a programmer can use for binary rewriting to inject arbitrary code at selected locations during runtime. It also includes the source code for instrumentation tools such as basic block profilers, cache simulators, and instruction trace generators. The DynamoRio (www.dynamorio.org) platform can perform program analysis of a running application, profiling, and binary rewriting or instrumentation. Both Pin and DynamoRio share the execution environment with the running application, and neither can handle applications involving multiple processes.

Network tools

Snort (www.snort.org) is a network intrusion protection system (NIPS) or network intrusion detection system (NIDS) that can detect buffer overflows, as well as attacks and probes such as stealth port scans, by performing content searching or matching and protocol analysis of real-time traffic. It is a solely signature-based system and can detect only attacks for which signatures are available. Bro (www.bro-ids.org) is a Unix-based NIDS that is not limited to identifying attacks based on signatures because it works at a higher level of abstraction. It uses vulnerability signatures and events to detect known attacks as well as patterns of failed connection attempts and connection service requests. Snort and Bro rely on manually generated rules and signature databases.

Although solutions and tools exist for flagging potential buffer-overflow vulnerabilities, they are inadequate because of the wide scope of the problem and each approach's inherent limitations. Extending and improving the existing defense methods for buffer-overflow exploits is imperative, especially the ability to handle new types of exploits.

Due to the diverse nature of the attacks, it is extremely difficult—if not impossible—to prefabricate methods for defending against them. Therefore, we suggest exploring methods that can defend against buffer exploits by acquiring knowledge from various sources in a dynamic and extensible way. Such sources might include expert specifications, analysis of code involved in new attacks, and trend analysis. We further suggest the integrated exploration of program analysis, pattern recognition, and data mining to establish such methods. 

Acknowledgment

This work is funded by the Centre for Strategic Infocomm Technologies, MINDEF Singapore.

References

1. C. Cowan et al., "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," *Proc. Foundations*

- Intrusion Tolerant Systems* [Organically Assured and Survivable Information Systems] (OASIS 03), IEEE CS, 2003, pp. 227-237.
2. J. Wilander and M. Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention," *Proc. 10th Network and Distributed System Security Symp.* (NDSS 03), Usenix, 2003, pp. 149-162.
 3. D. Nebenzahl, M. Sagiv, and A. Wool, "Install-Time Vaccination of Windows Executables to Defend against Stack Smashing Attacks," *IEEE Trans. Dependable and Secure Computing*, July-Sept. 2006, pp. 78-90.
 4. J. Xu et al., "Architecture Support for Defending against Buffer Overflow Attacks," *Proc. 2nd Workshop on Evaluating and Architecting System Dependability* (EASY 02), 2002; <http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.13.7372>.
 5. H. Ozdoganoglu et al., "SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address," *IEEE Trans. Computers*, Oct. 2006, pp. 1271-1285.
 6. J. Clause, W. Li, and A. Orso, "Dytan: A Generic Dynamic Taint Analysis Framework," *Proc. 2007 Int'l Symp. Software Testing and Analysis* (ISSTA 07), ACM, 2007, pp. 196-206.
 7. M. Rinard et al., "A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors)," *Proc. 20th Ann. Computer Security Applications Conf.* (ACSAC 04), IEEE CS, 2004, pp. 82-90.
 8. A. Smirnov and T. Chiueh, "DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks," *Proc. 12th Ann. Network and Distributed System Security Symp.* (NDSS 05), Internet Soc., 2005; www.isoc.org/isoc/conferences/ndss/05/proceedings/papers/dira.pdf.
 9. G. Novark, E.D. Berger, and B.G. Zorn, "Exterminator: Automatically Correcting Memory Errors with High Probability," *Proc. 2007 ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 07), ACM, 2007, pp. 1-11.
 10. D. Larochele and D. Evans, "Statically Detecting Likely Buffer Overflow Vulnerabilities," *Proc. 10th Usenix Security Symp.*, Usenix, 2001; www.usenix.org/events/sec01/full_papers/larochele/larochele.pdf.
 11. V. Ganapathy et al., "Buffer Overrun Detection Using Linear Programming and Static Analysis," *Proc. 10th ACM Conf. Computer and Comm. Security* (CCS 03), ACM, 2003, pp. 345-354.
 12. L. Wang, Q. Zhang, and P. Zhao, "Automated Detection of Code Vulnerabilities Based on Program Analysis and Model Checking," *Proc. 2008 8th IEEE Int'l Working Conf. Source Code Analysis and Manipulations* (SCAM 08), IEEE, 2008, pp. 165-173.
 13. W. Le and M.L. Soffa, "Marple: A Demand-Driven Path-Sensitive Buffer Overflow Detector," *Proc. 16th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.* (SIGSOFT 08/FSE-16), ACM, 2008, pp. 272-282.
 14. M. Cova et al., "Static Detection of Vulnerabilities in x86 Executables," *Proc. 22nd Ann. Computer Security Applications Conf.* (ACSAC 06), IEEE CS, 2006, pp. 269-278.
 15. A.K. Ghosh and T. O'Connor, "Analyzing Programs for Vulnerability to Buffer Overrun Attacks," *Proc. 21st Nat'l Information Systems Security Conf.* (NISS 98), 1998; www.ouah.org/ghosh98analyzing.pdf.
 16. P. Saxena et al., "Loop-Extended Symbolic Execution on Binary Programs," *Proc. 18th Int'l Symp. Software Testing and Analysis* (ISSTA 09), ACM, 2009, pp. 225-236.
 17. J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signal Generation of Exploits on Commodity Software," *Proc. 12th Ann. Network and Distributed System Security Symp.* (NDSS 05), Internet Soc., 2005; www.isoc.org/isoc/conferences/ndss/05/proceedings/papers/taintcheck.pdf.
 18. A. Smirnov, R. Lin, and T. Chiueh, "Automatic Patch Generation for Buffer Overflow Attacks," *Proc. 3rd Int'l Symp. Information Assurance and Security* (IAS 07), IEEE CS, 2007, pp. 165-170.
 19. M. Costa et al., "Vigilante: End-to-End Containment of Internet Worms," *Proc. 20th ACM Symp. Operating Systems Principles* (SOSP 05), ACM, 2005, pp. 133-147.
 20. X. Wang et al., "SigFree: A Signature-Free Buffer Overflow Attack Blocker," *Proc. 15th Usenix Security Symp.*, Usenix, 2006, pp. 225-240.

Bindu Madhavi Padmanabhuni is a PhD student in the School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore. Her research interests include software security, software analysis, and testing. Contact her at padm0010@ntu.edu.sg.

Hee Beng Kuan Tan is an associate professor in the Information Engineering Division at the School of Electrical and Electronic Engineering, Nanyang Technological University. His research interests include software security, software analysis, and testing. Tan received a PhD in computer science from the National University of Singapore. Contact him at ibktan@ntu.edu.sg.



Selected CS articles and columns are available for free at <http://ComputingNow.computer.org>.

COMPUTING THEN

Learn about computing history
and the people who shaped it.

<http://computingnow.computer.org/ct>