A Binary Rewriting Defense against Stack based Buffer Overflow Attacks

Manish Prasad Tzi-cker Chiueh

Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

{mprasad, chiueh}@cs.sunysb.edu

Abstract

Buffer overflow attack is the most common and arguably the most dangerous attack method used in Internet security breach incidents reported in the public literature. Various solutions have been developed to address the buffer overflow vulnerability problem in both research and commercial communities. Almost all the solutions that provide adequate protection against buffer overflow attacks are implemented as compiler extensions and hence require the source code of the programs being protected to be available so that they can be re-compiled. While this requirement is reasonable in many cases, there are scenarios in which it is not feasible, e.g., legacy applications that are purchased from an outside vendor. The work reported in this paper explores application of static binary translation to protect Internet software from buffer overflow attacks. Specifically, we use a binary rewriting approach to augment existing Win32/Intel Portable Executable (PE) binary programs with a return address defense (RAD) mechanism [1], which protects the integrity of the return address on the stack with a redundant copy. This paper presents the disassembly and instrumentation issues involved in static binary translation, how our tool achieves satisfactory disassembly precision in the presence of indirect branches, position-independent code sequences, hand crafted assembly code and arbitrary code/data mixing, and how it ensures safe binary instrumentation in most practical cases. The paper reports our experiences with this approach, based on results of applying the resulting prototype to rewriting several commercial grade Windows applications (Ftp server, Telnet Server, DNS server, DHCP server, Outlook Express, MS FrontPage, MS Publisher, Telnet, Ftp, Winhlp, Notepad, CL compiler, MS NetMeeting, MS PowerPoint, MS Access, etc.), as well as experimentation with published buffer overflow exploits.

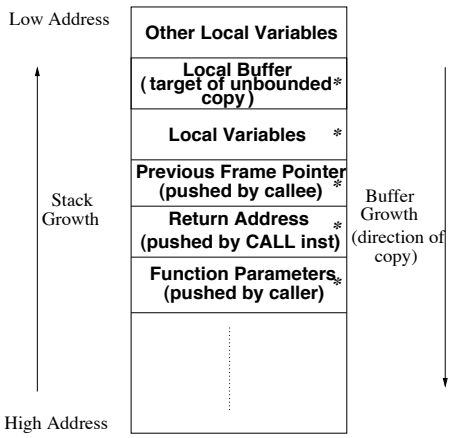

Figure 1. The typical stack layout if a function when it is called, and how some of the stack entries, including the return address could be corrupted by an unsafe copy operation.

1 Introduction

Buffer overflow attacks exploit a particular type of program weakness: lack of array/buffer bound check in the compiler or in the applications. Accordingly, the ideal solution to the buffer overflow vulnerability problem is to build a bound checking mechanism into the compiler, or to require applications to strictly follow a programming guideline that checks the bound of an array/buffer upon each access. Neither solution is considered practical at this point. A more promising approach is to transform a given application into a form that is immune from buffer overflow attack without requiring any modification to the compiler or the application itself.

To understand a buffer overflow attack, consider a typical stack layout when a function is called, as shown in Figure 1. Lack of bound checking during a buffer copy operation causes areas adjacent to the buffer (as shown by * in Figure 1) to be overwritten. A generic buffer overflow attack [2] involves exploiting such an unsafe copy to overwrite the return address on the stack with the address of a piece of malicious code, which is injected by the attacker and most likely reside also on the stack; when the `RET` instruction (which pops the return address from the stack) in the victim function is executed, program control is transferred to the injected malicious code.

A less common form of buffer overflow attack involves corrupting memory pointer variables on the stack instead of return addresses [18]. The requirements for such an attack to occur are:

1. A pointer variable p that is physically located on the stack after the buffer a[], to be overflowed,

2. An overflow bug that allows overwriting this pointer p by overflowing a[] (taking user-specified data as source), usually with the address of a GOT entry, which contains the address of a dynamic library function,

3. A copy function such as str(n)cpy/memcpy which takes *p as the destination and user-specified data as the source, without p being modified between overflow and copy,

4. A call to a common library function (like printf), the GOT entry of which is to be overwritten.

So leveraging the unsafe copy, the attacker overflows a[], thus overwriting p with the address of a GOT entry of a common library function, say printf(). By providing the address of the exploit code as input to the safe copy taking *p as the destination, the attacker has managed to corrupt the GOT entry of printf(). So any subsequent call to printf() would transfer control to the exploit code.

Another form of buffer overflow attack overwrites up to the old base pointer [19] on the stack with the address of malicious code, so that when the caller function returns, the control is transferred to the exploit code.

A seemingly non-stop stream of buffer overflow attacks [3, 4] has called for an effective and practical solution to protect application programs against such attacks on the Windows platform. Past approaches to protecting programs against buffer overflow vulnerabilities relied on compiler extensions, either to perform array bounds check or to prevent the return address on the stack from being overwritten. Although fairly successful in preventing most conventional buffer overflow attacks [2], these approaches require access to program source code. All known systems that take such an approach require the availability of the protected application's source code. While integrating software protection into a compiler is technically desirable, this approach exhibits several practical limitations. First, requiring access to source code makes it difficult to protect third-party legacy applications whose source code is unavailable for various reasons. Second, because modern software applications tend to be built on third-party libraries, access to the source code of these libraries again is unlikely. Finally, for those program segments that are written in assembly code directly, their high-level-language source code that is amenable to compiler analysis simply does not exist. The goal of this paper to describe our experiences and the extent of success that we have achieved in applying a combination of well known disassembly techniques to implement a binary rewriting solution that aims to provide the same level of protection as its compiler-based counterpart

There are two major technical challenges in applying binary rewriting to the buffer overflow attack problem. First, determine where to insert protection instructions, the boundary of each function in an input program needs to be clearly identified, which in turns requires an accurate disassembler that can correctly decode each instruction in an executable binary. Unfortunately, 100% disassembly accuracy is difficult because the problem of distinguishing code from data embedded into code regions is fundamentally undecidable. Second, even if the function boundaries are successfully identified, inserting protection code into a given binary without disturbing the addresses used in its existing instructions is itself non-trivial. The main problem here is that in many cases it is possible that the binary does not have enough spare space to hold a jump instruction to the inserted protection code, let alone to hold the protection code itself.

Section 2 surveys related work in the areas of static binary translation and buffer overflow defense. Section 3 discusses the design and implementation of our disassembly engine and the binary instrumentation issues with emphasis on the approaches we employ to ensure program safety and to preserve the semantics of input programs. Section 4 details the software architecture and implementation of the binary rewriting RAD prototype. Section 5 presents experimental results on the prototype's resistance to attacks, ability to preserve the semantics of applications, space cost and performance overheads. Finally Section 6 summarizes the main results of this work and charts out directions for future research.

2 Related Work

Among past efforts on binary rewriting, ATOM [5] and EEL [6] run on RISC architectures, where the disassembly problem is simplified due to uniform instruction size. Etch [7] is a tool for rewriting Win32/Intel PE executables primarily for optimization. LEEL [8] works on Linux/x86 binaries, albeit with limitations with respect to control flow analysis in presence of indirect control transfer instructions and arbitrary code/data mixing. UQBT [9] is an architecture independent static binary translation framework for migrating legacy applications across processor architectures. Galen Hunt's Detours [24], is a system for run-time binary interception of Win32 functions.

Most buffer overflow defense proposals involve compile-time analysis and transformation. Stackguard [10] and Microsoft Compiler Extension [11] place 'canary words' on the stack between the local variables and return address at the function prolog, and monitors the return address on the stack by checking the integrity of the 'canary word', at the epilog. Both are vulnerable to attacks based on corrupting old frame pointers [19] on the stack or local pointer vari-

ables [18]. Stackshield [12] and RAD [1] save a copy of the return address at the prolog and compare it with the return address on the stack at the epilog. Our binary rewriting implementation is based on this model of buffer overflow defense. Both are resilient to frame pointer based attacks [19] but vulnerable to memory pointer corruption [18] attacks. IBM's gcc extension [14] also does local variable reordering, placing pointer variables at lower addresses than buffers in addition to the above, offering some protection against memory pointer attacks [18], unless the unsafe copy is from higher to lower indices of the array. CASH [15] and others [16] perform array bounds check to prevent overflow of buffers. CASH achieves significant overhead reduction (4% overhead) by exploiting Intel segmentation hardware as compared to the others in this category, which typically incur very high overhead (70% to 140%).

Other proposed approaches to protect programs from buffer overflow attacks rely on run-time interception and checking. Lucent Bell Labs' Libsafe [20] intercepts unsafe library calls at run-time and performs bounds checking on the arguments e.g. for strcpy(), it would check the length of the source string and check it against the upper bound on the length of the destination string based on the current frame pointer value. Although it prevents the return address from being modified, it is possible to corrupt local pointer variables. Libverify [20] performs dynamic binary translation to perform return address check. However, we suspect that Libverify might incur high overhead, since it adds checking code and performs code instrumentation at run-time. A very recent example of applying optimized run-time interpretation to security problems is program shepherding [21], which is built on top of a dynamic optimization framework called RIO. Apart from offering advantages like complete transparency, it achieves significant overhead reduction as compared to what one would expect from an interpretation/emulation system using a variety of optimization techniques viz. traces and interpreted code caching.

To the best of our knowledge, the binary-rewriting RAD system described in this paper is the first attempt that employs static binary translation to protect existing binaries against buffer overflow attacks without requiring access to their source code.

3 Binary-Rewriting Return Address Defense

A successful binary rewriting RAD system requires identifying the boundary of every procedure in the input program and inserting a protection instruction sequence into every procedure without disturbing the input binary's internal referencing structure. The following two subsections discuss in more detail these two issues and their associated solutions.

3.1 Binary Disassembly

3.1.1 Disassembly Challenges

To accurately locate the procedure boundary, one needs to identify each instruction in the binary through a disassembler. There are two main classes of disassembly algorithms [22]. A linear sweep algorithm starts with the first byte in the code section and proceeds by decoding each byte, including any intermediate data byte, as code, until an illegal instruction is encountered. A recursive traversal algorithm starts at the program's main entry point and proceeds by following each branch instruction encountered in a depth-first or breadth-first manner, essentially a control flow analysis. Neither approach is 100% precise. The chief impediments to accurate disassembly are:

1. Data embedded in the code regions,
2. Variable instruction size,
3. Indirect branch instructions,
4. Functions without explicit `CALL` sites within the executable's code segment,
5. Position independent code (PIC) sequences, and
6. Hand crafted assembly code.

1) and 2) render the linear sweep algorithm less effective than ideal, whereas 3), 4) and 5) degrade the efficacy of the control flow analysis used in the recursive traversal algorithm.

Distinguishing code from data from a binary file is a fundamentally undecidable problem. Because the linear sweep algorithm decodes each byte as code as long as it looks like a legitimate code byte, it ends up interpreting many data bytes as instructions. The reason for this behavior is that in the Intel x86 instruction set 248 out of 256 possibilities can be a legitimate starting byte for an instruction, making it more likely to mistake data for instruction. The fact that the Intel x86 instruction set allows variable instruction size further aggravates the problem of code/data distinction. Consider the following example sequence of bytes:

```
0x0F 0x85 0xC0 0x0F  0x85 .....
```

If we consider 0x0F as a code byte then we'll end up with the following disassembly:

```
jne offset
```

On the other hand if we consider 0x0F as a data byte and `0x85` as a code byte, then we get something like:

```
0x0F    // data
test eax, eax
jne offset
```

Thus a single disassembly error could result in many subsequent bytes being interpreted incorrectly, with the extent of error potentially unbounded. In contrast, a fixed-instruction-size architecture exhibits a self-correcting property: an interpretation error for one instruction word does not propagate to the next instruction word.

The recursive traversal algorithms cannot obtain 100% accurate disassembly results, either, because it is difficult to construct a complete control flow of an input binary in the presence of indirect branch instructions such as `call/jmp reg32` (e.g. `call eax`) or `call/jmp m32` (e.g. `jmp dword[esp + xx]`). One solution to this problem is to perform additional data flow analysis such as inter-procedural slicing and/or constant propagation [23] to figure out at compile time the value of the register or memory location used in indirect control transfer instructions. Apart from being difficult to implement, such an approach tends to greatly increase the disassembly time and itself does not guarantee 100% accuracy.

Procedures for which no explicit call sites in the input program can be identified include exception or signal handlers, callback functions, which is rife in GUI applications, and procedures all calls to which are through indirect branch instructions. Because there is no identifiable call to these functions, they cannot be discovered through control flow analysis, and as a result may be misclassified as data. In practice, signal/exception handlers pose few problems because their entry points are included in the program header in some cases.

The addressing in position independent code (PIC) does not rely on any particular position in the program's address space. Thus PIC code and jump table never have absolute address references. Instead the references are in the form of offsets with respect to a base value that is known at run time, mostly through the eip value. For example,

```
10: call 10
15:
      :
25: pop eax   // gets the return addr
              // value 15 into eax
26: call dword[eax + 20]  // call foo
      :
      :
35: // foo
```

In this case, in spite of having explicit `CALL` sites, standard control flow analysis cannot discover the target location of the function foo().

Hand crafted assembly code makes it difficult to identify procedure boundaries because they do not necessarily follow the code conventions established by standard compilers. These conventions provide useful hints to resolve potential ambiguities. As an example of code convention violation, some assembly code programs jump from one function into another function without going through the latter's main entry point.

3.1.2 Disassembly Engine Implementation

Our disassembly engine is built on the x86 instruction set parsing and disassembly capabilities of an existing disassembler [13]. We use a combination of well-known disassembly techniques, viz. recursive traversal and linear sweep (described briefly in the previous subsection 3.1.1) and complement them with compiler-independent pattern matching heuristics. We assume that the data expected in a code section are typically dispatch tables (address bytes), strings and compiler alignment bytes. Since the goal of this project is insert protection code into every procedure of the input binary, we should identify as many code bytes as possible; otherwise the transformed binary may have security holes. However, we place maintenance of original program semantics at a higher priority than security, so whenever in doubt we mark bytes as data instead of code, thus avoiding unsafe binary instrumentation. The following is a step-by-step description of the disassembly process:

1. Identify potential address bytes for dispatch table discovery and strings. Dispatch tables typically contain code section addresses. Since we know the address range of the code section, we can mark any sequence of 4 bytes, which have a value that lies within this address range as 'potential address' bytes. Sequences of printable characters that have a certain minimum length and are terminated by a null character are marked as 'potential strings.'

2. Starting from the program's main entry point, which is obtained from the input binary's PE header [17], we perform a control flow analysis on the binary to traverse the paths of the program's control flow graph. All code bytes identified in this step are marked as 'definitely code' and all associated data bytes marked as 'definitely data,' We also identify targets of `CALL` instructions as function entry points and targets of conditional and unconditional jump instructions as jump targets. Since this step can distinguish data from code with 100% accuracy, it overrides analysis results from other steps whenever there is a conflict. For example, the following byte sequence will be identified as a `call` instruction because the result from Step (2) overrides that of Step (1).

   ```
   Identified as instruction
   'call dword[0x30001344]' in Step (2)
   <-------------------------->
   0xFF 0x15 0x44 0x13 0x00 0x30
             <----------------->
             Identified as 'potential
             address' (0x30001344) in
             Step (1)
   ```

3. To identify the entry points of potential callback functions, for which there are no explicit call sites, we look for instruction sequences such as:

   ```
   push imm32
   mov reg32, imm32
   ```

   Typically the target address of a callback function is usually passed as an argument to some function, with which the callback function is registered. Such an argument could be passed through the stack as an immediate value (push imm32) or through a register, which contains the address value (mov reg32, imm32; push reg32). If the byte at imm32

has not been identified as a 'potential address' or a 'potential string' in Step (1), and if it looks like a legitimate instruction starting byte, we consider it as a function entry point (although despite being a legitimate code byte it may not actually be a function entry point) and proceed disassembling the subsequent bytes as instructions.

4. To identify other types of functions for which there are no explicit call sites, we next look for bytes in the code section that have not been identified as code or data yet. Every time such a byte is located, we start instruction parsing if it looks like a legitimate instruction starting byte. In both Steps (3) and (4), the point where such instruction parsing begins is called a 'reset point'. Instruction parsing continues until an unconditional branch instruction (`ret` or `jmp`) is encountered. If the result of an instruction parsing procedure is inconsistent with any previously identified byte or leads to an illegal instruction byte, the result since the reset point is revoked and all the bytes from the 'reset point' to the current position are marked as data, thus avoiding any potentially unsafe binary rewriting. After an unconditional branch we look for the next suitable 'reset point' to start the next instruction parsing attempt.
5. Because any sequence of instruction bytes should end with an unconditional branch instruction (`jmp` or `ret`), we look for code sequences that end without an unconditional branch (`jmp` or `ret`) instruction and mark such code sequences as data. This check provides a final line of defense to eliminate any potentially incorrectly identified instruction sequences. For example, in the following code sequence, Bytes 1 to 3 will be marked as data.

```
1: mov eax, ebx
3: push eax
4: data
```

   Also, the byte next to an unconditional branch has to be either a data byte or if it is a code byte, it must be a branch target (as the previous instruction, being an unconditional branch, doesn't fall through). Therefore, in the case that the byte next to an unconditional branch is a code byte and has not been marked as a function entry point or a jump target, we mark it as a function entry point, even though they could just as well be targets of a branch instruction inside the same function.

   The motivation behind this "over-identification" of functions, as seen in steps 3) and 5) will be explained in subsection 3.3.

3.2 Binary Instrumentation

Because it is not always possible to derive the high-level control flow of an input binary, the process of inserting additional code to counter buffer overflow attacks must proceed in a way that does not disturb the memory references used in the instructions of the binary program that is to be protected.

3.2.1 Where to Insert RAD Code

The additional code required by RAD [1] involves

- Saving a copy of the return address on the stack in the return address repository (RAR) at the function prolog, and
- Checking the return address on the stack with the saved copy in the RAR at the function epilog, popping it off the RAR in the event we have a match, or flag an exception otherwise.

Instead of adding function prologs and epilogs to every function, we choose to do so only for 'interesting' functions, which are functions that contain a sequence of instructions for stack frame allocation and deallocation for local variables. A function without local variables could never be vulnerable to a stack based buffer overflow.

3.2.2 How to Insert RAD Code

So as to not disturb the original binary's address space, we choose to create a separate new code section, not present in the original PE binary (information regarding the PE format is in [25]), appended to the end of the original binary to hold the additional prolog and epilog code for each function. Moreover this new section, mapped to a non-interfering portion of the address space, will be set as read-only. Thus neither the RAD code is corrupted by the application nor is the application corrupted by the RAD code. To redirect control to the inserted code at a function's prolog and epilog, we need to replace some instructions at the function prolog and epilog with a `JMP` to the corresponding RAD code. When such an instrumented function is invoked, the `JMP` instruction, which replaces the prolog, transfers control first to the RAD prolog code, then executes the original prolog instructions and then jumps back to the original function to continue execution from the instruction immediately after the original function prolog. Epilog instructions are replaced in a similar manner. However, the execution proceeds first with a `JMP` to the epilog code in the new section, first executing the original epilog instructions until the RET, then the RAD epilog checking code and then return if there are no problems. Because the size of an unconditional `JMP` instruction is 5 bytes, we need at least 5 bytes worth of instruction space to accommodate a `JMP` instruction. Instructions that are target of existing branch instructions cannot be replaced.

A function prolog, which needs to allocate stack space for local variables, typically comprises 3 instructions :

```
1.    push ebp  // save old frame ptr
                // (1 byte instruction)
2.    mov ebp, esp // set the top of
                   // the stack as the
                   // current frame ptr
                   // (2 byte inst)
3.    sub esp, x  // allocate x bytes on
                  // the stack for local
                  // variables (3 to 6
                  // byte instruction)
        or
      add esp, -x
```

Alternatively it could also be done using the enter instruction, however most compilers do not use enter for stack frame allocation. Thus, an 'interesting' function prolog includes at least 6 bytes

worth of instructions. Hence, we can comfortably instrument an 'interesting' function prolog to redirect control to the RAD prolog code using a 5-byte `JMP` instruction. On the other hand, a typical stack frame deallocation instruction sequence looks like one of the following three cases:

```
1.    add esp, x    // dealloc. stack
                    // space, x bytes
                    // were allocated
                    // (3-6 byte inst)
      pop ebp       // restore caller's
                    // frame ptr (1 byte)
      ret           // return (1 byte)
2.    mov esp, ebp  // dealloc. stack
                    // space, any
                    // number of bytes
                    // allocated on the
                    // stack (2 byte
    // instruction)
      pop ebp       // restore caller's
                    // frame ptr (1
                    // byte)
      ret           // return (1 byte)
3.    leave         // dealloc. stack
                    // frame & restores
                    // old frame ptr
    // (1 byte)
      ret           // return (1 byte)
```

From 2) and 3), we see that stack frame deallocation could be done with 2 to 4 bytes worth of instructions. So we need to replace some more instructions in addition to the stack frame deallocation instructions to hold a `JMP` instruction. In most cases, we do find enough space this way. However, it is possible that the first instruction of the stack frame deallocation sequence is a jump target, e.g.:

```
    jne x
      :
x:  leave
    ret
```

In this case, if we replace instructions prior to leave, then the jump target x would be disturbed. From our experiences, the scenario of not being able to find 5 bytes worth of instructions at a function's epilog does occur in practice but is relatively rare. For such a situation to occur in practice, two conditions need to be met:

a) Most development environments on Windows, by default, set certain compilation options which generate calls to stack checking code, prior to stack frame deallocation, to check for adherence to certain calling conventions (which basically dictate caller and callee duties as regards function frame initialization and cleanup). Calling convention adherence check is desirable because of functions being called using function pointers and calls to library functions. If we disable these options the compiler wont generate these stack checking calls and thus will not generate extra bytes prior to stack frame deallocation.

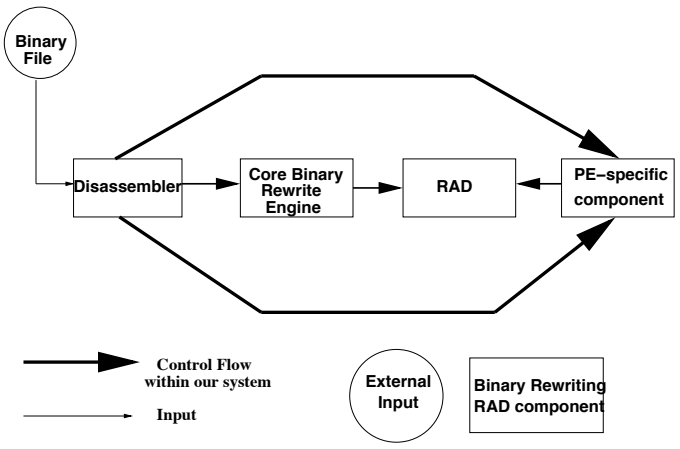

Figure 2. The software architecture of the binary-rewriting RAD prototype, which consists of a disassembler, a core binary rewriting engine, a RAD component, and a PE component.

b) There should be a high level code sequence like:

```
      goto label;
          :
          :
  label:
      return;
```

So in such rare scenarios (our experiments show typically 0.03 - 3% of all functions, sec. 5.2.2, table 9), we use a simple although expensive approach to solve this problem. When not enough instructions are available, we replace the first byte of the instruction prior to `ret` with an `int 3` (breakpoint interrupt) instruction, which corresponds to a software interrupt, and install a corresponding exception handler. When an `int 3` instruction is executed, it generates a Debugger Breakpoint Exception, and the handler gains control to perform return address check. Because this exception handler is executing the user space, control transfer to our handler is similar to an intra-privilege level far call, which means that there is no stack switching and the exception handler can access the return address on the stack. For details regarding how the stack evolves during the execution of a software interrupt handler, please refer to [17]. The reason why we chose the debugger breakpoint exception is that this exception is not used normally unless the program is being debugged. However, while being debugged under a debugger, the control is transferred to the debugger when an `int 3` instruction is executed, and our exception handler will not executed.

4 Prototype Implementation

4.1 Software Architecture

The binary-rewriting RAD tool comprises the following logical components: a disassembler, a core binary rewrite engine, a

RAD component, and a PE component. The disassembler functions in two main phases. In the first phase, it performs code/data and branch target identification of all bytes in the code section, and in the second, it outputs the assembly instructions starting from the first byte in the code section. The core binary rewrite engine, independent of the binary format, hooks into the disassembler in the second phase to gain control at every instruction processed to look for 'interesting' function prologs and epilogs to instrument. This component handles all the issues involved in adding instrumentation code outlined in the section 3.2. Since the instructions that make up an 'interesting' pattern need not be contiguous, this component maintains a circular window of five instructions (current instruction and four previous ones), which is flushed whenever a branch target is encountered, so that we don't run over any jump target. The engine attempts to identify 'interesting' patterns in the window every time a new instruction is added. All the RAD code and its associated data are added to a new section at the end of the input binary. The RAD component implements the Return Address Defense mechanism. The prolog stub saves the return address on the stack to the RAR and the epilog stub keeps popping the RAR stack till it finds the return address currently on the top of the stack or till the RAR is empty in which case it flags an exception. This repeated popping ensures that the return from any of the caller's ancestors (from the current call stack), does not generate false positives. This scenario occurs in case of setjmp(), longjmp() as well as compiler optimizations which cause functions to return straight to the caller's caller if the first return is to the caller's return section. The PE component initializes the binary rewrite process by adding a new section header in the section header table and setting up its fields appropriately; it also aligns the new section (called the .RAD section) that holds the RAD code where it should be loaded at run time (by rounding off to the page boundary after the end of the previous section) and where it should be stored in the binary file (by rounded off to the file alignment boundary after the end of the previous section).

4.2 RAR Initialization

The .RAD section is set as read/write/executable. It needs to be writable since the RAR is also a part of this section in addition to the RAD code. However, At run time the non-RAR part of this section needs to be set to read-only, through a Win32 API call VirtualProtect(). This is to create mine-zones on both the sides of the RAR to prevent attackers from overflowing the RAR. The key issue here is how to locate the entry point of VirtualProtect(). There are several cases to consider. First, it is possible that the input program also uses VirtualProtect() for some other purpose and thus has loaded it into the address space by the time the PE component takes control. Second, it is possible that the input program does not need VirtualProtect(), and therefore the PE component needs to load this function into the address space itself.

For the first case, two PE headers of interest here are the Import Address Table (IAT) and the Import Name Table (INT). The two tables can be reached from the import directory, whose location in turn is obtained from the DataDirectory array in the PE Optional-Header. Each entry in the IAT, at run time, contains the address of an imported function and the on-disk binary file. For each IAT entry there is a corresponding entry in the INT, which points to the name of the corresponding imported function. At load time, the loader overwrites the IAT entries with the virtual addresses where the corresponding functions are mapped. To locate the entry point of VirtualProtect(), we look up the INT by its name and retrieve the index of the associated IAT entry if there is a match. If VirtualProtect() is already imported by the input program, then its entry point is readily available from its IAT entry. If there is a match in the INT but the corresponding IAT entry is empty, then we need to dynamically resolve the entry point of VirtualProtect() by calling GetModuleHandle(), which gets the base address of the DLL containing the API function in question, and then calling GetProcAddress(), which gets the address of the desired API function from the export directory of its containing DLL. The entry points of GetModuleHandle() and GetProcAddress() themselves are obtained from the IAT through their containing DLL, kernel32.dll. In the case that none of these API functions are themselves imported, which is quite rare, then the PE component needs to emulate the operation of GetModuleHandle() and GetProcAddress(). This emulation idea is derived from an undocumented virus code and is based on the following observation: At the program entry point, the top of the stack contains a return address, which points to somewhere inside the function CreateProcess(), which in turn belongs to the DLL kernel32.dll. With this address, one can scan through the memory until where kernel32.dll is mapped is found. Once the base address of kernel32.dll is known, the entry points of GetModuleHandle() and GetProcAddress() are available through the DLL's export table, and in turn the entry point of any API function can be identified through these two functions.

Finally, the entry point of the executable in the PE header should be changed to the new initialization code, which locates the entry point of VirtualProtect(), and calls it to protect the two surrounding pages of the RAR.

4.3 Limitations

4.3.1 Security Weaknesses Due to Disassembly Limitations

Two aspects of disassembly that relate to sources of false positives and security loopholes are:

- Functions not covered by our disassembly engine
- "Over-identification" of functions

Let's look at each of these aspects and evaluate when and how these could result in false positives or missed attacks. There is a tradeoff between security and program correctness. While we make every effort to seal all known security holes, preserving original program semantics is considered as a more important goal than achieving perfect security.

Missed Functions

These (if any) are mainly callback functions (without an explicit `CALL` within the code section) and/or functions invoked using Position-Independent Code (PIC) sequences (wherein there would be no absolute address references to a function within the code section). These are not covered by pure control flow analysis. Despite this, typically, only a certain fraction of such functions get

missed out. The following "representative" scenarios should make this clear. When functions are missed by the control flow analysis step, they could be:

a) identified fully as code or

b) partly/fully misidentified as data or

c) aligned with data (preceding the actual function entry) and misidentified as code.

Considering a), typically we would have the unidentified function entry point preceded either by data or by an unconditional branch instruction from the previous function, in either of the cases, we would indeed mark the function entry point (last step (5) of disassembly engine sec. 3.1.2). If a function is fully identified as data, then we miss the function altogether, and this might lead to a potential security loop- hole, if that particular function has a buffer overflow vulnerability. Considering part misidentification of code as data, say, the start of such a function is misidentified as data, then we might miss out on an interesting prologue, and if the end of the function is identified as data, then we might miss an interesting epilog, and hence, either cases result in an unprotected return, which is a recipe for a potential security loophole. When a middle part of a function is misidentified as data, then the function is divided into two; so all returns within this function beyond this point would be treated as a part of an uninteresting function, and hence are left uninstrumented and could miss an attack.

When data preceding the entry point of such a function aligns properly with the code bytes to form a legitimate instruction sequence, an originally interesting prologue could become uninteresting, thus exposing an attack opportunity. In all the cases presented so far, however, program semantics are not jeopardized. But if in c), data misidentified as code turns an uninteresting function prolog to an interesting one, it might generate a false positive, if the epilog happens to be interesting. Another false positive scenario is if the function entry point is preceded by some data and the first identified code byte happens to be a jump target (happens with inter-procedural jumps), in which case the two functions get merged into one. However, inter-procedural jumps occur only in handcrafted assembly or as in setjmp, longjmp cases.

Apart from functions, jump targets reached by PIC jump tables could be missed. This could affect program correctness, if these targets happen to be within instrumented prologs or epilogs, a very unlikely scenario, though.

Falsely identified functions

Functions with multiple entry points are treated as two separate functions. Targets of PIC jump tables, which cannot be discovered statically could get marked as function entry points, if they lie immediately after an unconditional branch or a sequence of data bytes (last step (5) of disassembly engine sec. 3.1.2). Code section addresses which appear as immediate (`imm32`) operands to `mov r32, imm32` or `push imm32`, could be identified as function entry points even if they are targets of an indirect jump (non-PIC jump table targets are, however, treated specially and identified).

Function boundary identification helps prevent scenarios where the prolog is instrumented, but the epilog is not and vice versa. Since the latter case could cause false positives (since the epilog checking code would be trying to find a match for the return address on the stack, but since it was never saved (no prolog instrumentation code), it won't find it in the RAR, thus flagging a false exception), we want to avoid that altogether, which can be achieved by over-identification of functions. Over-identification, however could result in a function having an instrumented prolog, but an uninstrumented epilog. Such a function, if called too frequently in a manner that it exits from an uninstrumented epilog, then the RAR will eventually overflow, since there is no code to pop the return address off the RAR. Another potential problem due to over-identification is missed attacks. If over-identification causes an "entry point" to be inserted within a function body then the single function gets divided into two. Here the "second" function won't have an interesting prologue, hence all subsequent returns in this function will be missed. However, "over-identification" never jeopardizes program correctness unless, of course, an entire chunk of data misidentified as code forms a function, with both interesting prolog and epilog, which is an extremely unlikely scenario.

In summary, PIC indirect branches and callback functions could cause some security loopholes in the input programs to be unprotected. Empirical results show that indirect branches typically are 5-8 % of all branch instructions (Section 5.2.1, Table 5). Only a fraction of this (if at all any) could possibly result in a missed attack. As for false positives, they could arise due to

a) Hand crafted assembly code, mostly with inter-procedural jumps and/or entry and exit points in different functions. Here's an example of such a case that we observed after rewriting the application Microsoft Access:

```
Fn1:  // no 'interesting' prolog
        :
      jne label
        :
      ret // no 'interesting' epilog

Fn2:  // 'interesting prolog'
        :
label:
        :
      ret // 'interesting' epilog
```

In this case, the control jumps from Fn1 (whose prolog is not instrumented, meaning its return address is not saved in the RAR) to label, which is in Fn2 and exit from Fn2 (whose epilog is instrumented, meaning a return address check is done). The RAD epilog of Fn2 will flag an exception, since it cannot find the on-stack return address in the RAR, thus a false positive. A solution to avoid such false positives could be to assign an ID to every function, and keep the function ID information as the part of the RAD bookkeeping and checking operation. With this additional piece of information, a RAD epilog check only proceeds when its function ID matches the function ID of the top of stack on the RAR; otherwise the epilog check should does nothing. The current binary-rewriting RAD prototype does not yet support this mechanism.

b) Data misidentified as code which looks exactly like an interesting prolog, or an entire chunk of data which look like an interesting function.

All of the above are rather uncommon scenarios.

4.3.2 Potential buffer overflow attacks due to limitations of RAD

As in RAD [1], the current binary-rewriting RAD prototype can protect applications from any kind of buffer overflow attack that corrupts the return address on the stack. Thus it can resist conventional stack smashing attacks and frame pointer based attacks [19]. However, it cannot prevent memory pointer corruption attacks [18], which do not affect the return address in any way. They simply modify the contents of the import table (GOT or IAT), and therefore it is impossible for RAD to detect them. Fortunately, no actual network security breach incidents that are based on this type of attacks have been reported.

4.3.3 Multithreaded Applications

The current implementation doesn't handle multithreaded applications. An idea to implement the solution for multithreaded applications, comes from [26]. We can access the Thread Information Block (TIB) structure using the FS segment register. Code generated by compilers to set up exception handlers and to allocate storage for thread local variables, typically reveal this use of the FS register. The TIB contains an array of slots for thread local storage. What we could do is have a separate RAR space for each thread (taking care that RAR spaces of two threads donŕ bump in to each other), and store the address of the RAR in one of the thread local storage slots, which can be used by the RAD prolog and epilog code, to figure out which RAR to work with. However, the use of the FS register, although a well-known fact in the Windows world, still falls into the category of undocumented information. There would probably be Win32 API functions, which do something like this, however the cost of invoking an API call at every RAD prolog and epilog would be prohibitive.

4.3.4 Self-Modifying Code

Self-modifying code, like those missed functions due to indirect branches, makes control flow analysis difficult. Moreover, if a piece of code is added only at run-time to the heap, there is no way RAD can add checks to it.

5 Experimental Results

To validate the correctness of the binary-rewriting RAD prototype, we need to verify that the RAD code is injected into appropriate places in the input binary AND the RAD code does protect the input binary from buffer overflow attacks in a way that does not incur significant space overhead or run-time performance cost. In the following subsections, we present results that show that the current binary-rewriting RAD prototype does do a reasonable job in disassembly accuracy and low-overhead protection against buffer overflow attacks.

Step	Size
Return Address Repository	16 Kbytes
Exception Handler	130 bytes
Installing Exception Handler	19 bytes
Set up RAD mine-zones	55 bytes
Search for VirtualProtect()	371 bytes
Total	**16.2 - 16.6 KBytes**

Table 1. *The constant space overhead of binary-rewriting RAD. The last row corresponds to the step that searches the kernel32.dll export table for the entry point of VirtualProtect().*

5.1 Micro-Benchmark Results

To establish the baseline performance for the binary-rewriting RAD prototype, we apply it to a set of synthetic programs and measure its space and performance overhead. Table 1 shows the constant space overhead associated with binary-rewriting RAD, which excludes the per-function prolog and epilog RAD code. Every instrumented function needs a prolog and epilog checking code, which take 38 and 41 bytes, respectively.

We then measure the performance overhead of an instrumented function due to its prolog and epilog RAD code. Depending on whether an epilog RAD code is triggered by a jump instruction or by a software exception handler, the measured performance overhead is different. We tested three different instrumented functions:

- void fn() that does nothing and invokes prolog and epilog RAD code through a jump instruction
- void fn() that does nothing and invokes prolog RAD code through a jump and epilog RAD code through a software exception
- void fn() that does some amount of computation (incrementing a variable 25000 times), without making any other function calls

The performance penalty of the binary-rewriting RAD prototype is defined as:

$$Penalty = \frac{fn's\ additional\ cost\ due\ to\ RAD}{fn's\ original\ cost}$$

and was measured using the Pentium performance counter, which has a resolution of 2 nsec.

For the do-nothing test function case, the overhead of RAD is 34.25%, which is higher than expected, considering that both the prolog and epilog RAD code size is just about 9 to 11 instructions, each of which is such a simple instruction as '`push reg32`', '`pop reg32`', '`mov reg32, mem32`', '`cmp reg32, mem32`', '`add mem32, imm32`' '`sub reg32, imm32`' etc., none of which appear to be costly. We believe this performance overhead is due to additional instruction cache misses that arise because the code region of the test function is separate from that of its prolog and epilog RAD code. If a function's epilog does not contain enough space to hold a jump instruction, the epilog RAD code

Test Function	Cycle Counts - Original	Cycle Counts - RAD	Relative Penalty
Null function	292	392	34.25%
Null function + epilog	271	641	136.53%
Incrementing function	350,425	350,722	0.085%

Table 2. *Per-function performance overhead due to the RAD code injected into an instrumented function. The* **Null function + epilog** *case is the same as the* **Null function** *case except its epilog is invoked through a software interrupt. The* **Incrementing function** *case corresponds to a function that increments a variable 25000 times. All measurements are in terms of Pentium cycle counts.*

is implemented inside an exception handler. The additional performance overhead due to exception delivery and return, as compared to two jump instructions, is almost quadrupled, as shown in the second test function case of Table 2.When the test function is doing something more computation-intensive, as in the third test function case, the relative performance overhead of RAD immediately becomes negligible. Compared with the original RAD system [1], which works on source code only, binary-rewriting RAD performs better in all three cases, because its prolog and epilog code is implemented in assembly and is thus more efficient. This result is somewhat surprising as the original RAD system places per-function prolog and epilog code together with the associated function, rather than in a separate code region, and therefore does not incur additional instruction cache miss penalty.

5.2 Macro-Benchmark Results

We experimented with a wide variety of commercial grade Windows applications, including BIND DNS server, DHCP server, a third-party FTP server, Microsoft Telnet Server, MS FrontPage, MS Publisher, MS PowerPoint, MS Access, Outlook Express, CL compiler, MSDEV.EXE (Visual C++ development environment), Windows Help (Winhlp), and Notepad. After rewriting, all the above programs behave exactly the same as before, except MS Access, which generated a false positive due to hand crafted assembly code (described in Section 4.3), and the third-party FTP server, which has an internal exception handler that conflicts with the debugger exception handler that binary-rewriting RAD installs. The initial experiences collected from running the binary-rewriting RAD prototype against a wide array of regular desktop applications and Internet servers, which are the prime targets for buffer overflow attacks, convinced us that this prototype is sufficiently mature to preserve the program semantics of complex production-grade applications while providing them with protection against buffer overflow attacks. Of course, more exhaustive tests are required to be absolutely sure about the accuracy of disassembly and the protection strength of RAD.

5.2.1 Disassembly Accuracy

The binary-rewriting RAD prototype uses control flow analysis and a set of other heuristics to distinguish between code from embedded data. In general, control flow analysis is quite effective in identifying the code regions for non-interactive applications, which usually do not have many call-back functions. However, for interactive GUI applications, such as those in Microsoft Office suite, control-flow analysis alone is not quite as effective because of the hidden call-back functions. Therefore these applications represent the most challenging test programs for a disassembler. Table 3 shows the disassembly accuracy of three such programs, MS Powerpoint, MS Frontpage, and MS Publisher. The disassembly accuracy of all these programs is above 99%. The way we measure disassembly accuracy is to manually inspect the resulting assembly code and determine whether the instructions look "reasonable." From our experiences, instructions that are disassembled from data tend to appear out of place and thus can be easily detected.

Since the manual inspection method used above may not seem rigorous enough, to further verify our disassembly results, we experimented with certain Cygwin ported Unix applications (with available sources) on Windows, compiled with gcc's profiling options and then analysed them offline with gprof.

Thus the % of both missed functions and unprotected returns in interesting functions appear to be reasonable. The missed functions in Apache were typically functions without any absolute address references in the code section, which were invoked through a table of function pointers to which the addresses of those functions were assigned statically. The table, being a static array variable, is located in the .data section and so were the function addresses. The static call graph generated by gprof also shows the parents of these functions as 'unidentified'.

Because the results obtained from control flow analysis is guaranteed to be correct, it is useful to measure the percentage of instructions that can be identified through control flow analysis, which gives an indication of how useful other heuristics are in identifying instructions, especially for GUI-intensive interactive applications. Table 5 shows the total number of instruction bytes in each test application and the percentage of them that control flow analysis can successfully detect. As expected, for non-interactive applications, which rarely use any call-back functions, control flow analysis can achieve a very high detection accuracy, more than 97%. However, for interactive applications, the percentage is around 80%. The difference between the coverage percentages in Table 3 and 4 for the three programs, MS Powerpoint, MS Frontpage, and MS Publisher, represent the contribution of the pattern-based heuristics that binary-rewriting RAD employs to the total code region coverage.

Finally, because control flow analysis plays such an important role in the disassembly process, it is instructive to investigate deeper why it cannot detect all the instructions in the program. Other than functions that do not have explicit call sites,

Application	Code section size	No. of incorrectly decoded bytes (approximation)	Accuracy
MS PowerPoint	4.059 MB	2500	99.93%
MS Publisher	2.314 MB	50	99.99%
MS FrontPage	983 KB	900	99.91%

Table 3. *Disassembly accuracy achieved as measured through manual inspection of the resulting assembly code. Higher accuracy means that more bytes are successfully disassembled.*

Application	No. of functions (source code)	No. of functions (disassembly)	Miss %	No. of returns unprotected	% of returns unprotected
Gzip	234	234 (80)	0	2	0.85
Wget	626	626 (140)	0	3	0.48
Apache	1191	1159 (350)	2.69	38	3.19
Whois	148	148(15)	0	0	0
OpenSSL	2820	2812(780)	0.283	7	0.248

Table 4. *Evaluation of disassembly results by comparison with original program sources. The numbers in the parenthesis on the third column represent the number of "over-identified" functions.*

indirect branch instructions are the main culprit. We measure the percentage of indirect control branch instructions in the test applications and the results are shown in Table 5. Again, interactive applications such as MS Powerpoint and Access tend to have a higher percentage of indirect branches than others, which reflects the event-driven programming style of these applications, and correspondingly a more extensive use of function pointers and switch statements.

In summary, our disassembly results appear to be better than the previous best reported in the literature [22], which claims 99.9% precision using binaries with relocation information, but most of their experiments were on smaller programs, all of which were plain command line programs without any GUI callback functions, which makes disassembly is tougher. Furthermore, the presence of symbol table information in binaries (possibly inadvertently) eliminates problems regarding function boundary identification. However, there is a question of the symbol table format. It could either be the generic COFF symbol table, supported by the PE/COFF binary formats, or it could be a compiler specific format, like the VC++ .pdb. Apparently, compilers tend to favor their proprietary formats for symbol table over the generic COFF format. This is evident since the default compilation options for generating debug information do not produce COFF symbol tables, and generate proprietary symbol tables instead.

5.2.2 Run-Time Overhead

An important consideration in the design of RAD is the minimization of performance overhead due to per-function prolog and epilog RAD code. The relative performance overhead of RAD with respect to a test application is defined as

$$\frac{Execution\,time\,with\,RAD\;-\;Execution\,Time\,w/o\,RAD}{Program\,Execution\,Time\,without\,RAD}$$

The results are in Table 6, which shows the run-time performance overhead of binary-rewriting RAD for typical Internet applications is quite small, around 1%. The space overhead of binary-rewriting RAD for real applications is also quite reasonable, as shown in Table 7. The highest percentage is still smaller than 35%. Both results demonstrate that the overhead of binary-rewriting RAD is quite reasonable for practical applications, given the additional protection it provides.

Because the cost of invoking an epilog RAD code through an exception handler is around four times as expensive as that through a jump instruction, it is important to find out how frequent epilog RAD code is invoked through an exception handling mechanism. If it occurs frequently, then perhaps a more sophisticated mechanism needs to be developed. Table 7 also shows the percentage of functions in the test applications whose epilog RAD code is triggered via an exception handler. The statistics in Table 7 show that the percentage of functions that do not have enough instruction space for a jump instruction is fairly low, less than 3%, which justifies our design decision of using this expensive solution in these infrequent cases. Please note that, these are results of static analysis. It is possible that, at run-time one of these functions get invoked 50% of the times, in which case the performance might get seriously hit. While it is possible to instrument binaries to report the % of functions called at run-time which need the use of the `INT 3` software interrupt, we are not clear if that would say much, since at the end of the day, we can still just say that, since the % of such functions (from our experiments) is typically 0.03% to 2.5%, probabilistically the % of such functions among those called at run-time would be of a similar order.

Application	Percentage of code section covered by control flow analysis	Percentage of indirect branch instructions
WFtpd (Ftp server)	97.13%	5.81%
BIND (DNS server)	97.32%	5.42%
MS Access	84.57%	8.29%
Notepad	97.54%	1.73%
MS Powerpoint	80.11%	7.54%
Windows Help	99.67%	1.41%
MS FrontPage	87.20%	8.98%
MS Publisher	93.86%	8.94%

Table 5. *Column 2 shows the percentage of a program's code section bytes that is detected purely through control flow analysis. Column 3 shows the percentage of indirect branch instructions among all the branch instructions.* `RET` *instructions are not included in this count.*

Application	Original execution time (msec)	Binary RAD execution time (msec)	% Overhead
BIND	122.56	123.85	1.05%
DHCP server	122	123.5	1.23%
PowerPoint	145	150	3.44%
Outlook Express	138.2	140	1.29%

Table 6. *Whole program performance overhead due to the insertion of binary-rewriting RAD code. For BIND, the response time measurement is averaged over 10 queries issued using the client program* `dig.exe`*. For the DHCP server, the measurement is the startup and initialization time averaged over 6 runs. For PowerPoint, the measurement is the time taken to render a 90Kbytepresentation averaged over 6 runs. For Outlook Express, the measurement is the startup and initialization time averaged over 6 runs.*

5.3 Resilience to Buffer Overflow Attacks

We simulated a buffer overflow exploit based on [2] on a test program with buffer overflow vulnerability. With binary-rewriting RAD, it successfully detects and stops the attack.

6 Conclusions and Future Work

We have presented a buffer overflow defense mechanism using static binary translation based on the RAD [1] model. To the best of our knowledge, this is the first work reported in the open literature that applies static binary translation technology to a concrete application security problem. While a robust binary rewriting infrastructure such as tools like Etch [7] does exist, published papers on these systems never document in detail the design and implementation issues involved, the solutions adopted to address them and their effectiveness in a quantitative manner. We believe that this paper exhaustively covers most binary translation issues in substantial depth and detail and presents a comprehensive set of experimental results to demonstrate the efficacy of the design decisions we have made. Finally, the resulting binary-rewriting RAD system achieves qualified success as an important tool to protect legacy applications whose source code is not available against buffer overflow attacks, and thus significantly broadens the applicability of buffer overflow defense mechanisms developed in the research literature. Although, it may not achieve the stated goal of providing the same level of protection as its compiler-based counterpart, in a few cases, it is primarily due to a fundamental deficiency, one that none of the known works in the binary translation literature have done better with, as far as we can tell.

Currently, we are exploring more robust and foolproof fallback mechanisms to deal with scenarios of incorrect disassembly and lack of sufficient space for 'in place' translation. As an immediate next step, we intend to experiment our binary translation engine with Dynamically Linked Libraries (DLLs), since a major chunk of Windows services are implemented as DLLs. Finally, we aim to apply the lessons from exploring static binary translation techniques to build copy- and tamper-resistant software.

References

[1] Tzi-cker Chiueh and Fu-hau Hsu, "RAD: A compile time solution for buffer overflow attacks", ICDCS 2001

[2] Aleph One, "Smashing the stack for fun and profit", Phrack Magazine 7 (49), November 1996

[3] David Litchfield, "Windows NT buffer overruns" RAS: http://community.coresdi.com/ juliano/mnemonix-

Application	Percentage Increase in size	Percentage of functions that need `INT 3` software interrupt
WFtpd (Ftp server)	34.06%	2.57%
BIND (DNS server)	32.65%	0.00%
MS Access	11.29%	2.61%
MS Powerpoint	9.74%	0.83%
Windows Help	32.79%	0.098%
MS FrontPage	16.45%	0.031%
MS Publisher	10.84%	1.58%

Table 7. *Column 2 shows the space overhead of binary-rewriting RAD for different test applications in terms of percentage increase in size of the executable file after rewriting. Column 3 shows the percentage of functions among those identified that need to invoke RAD epilog code through the* `INT 3` *handler*

rasbo.htm Winhlp32: http://community.core-sdi.com/ juliano/mnemonix-whlpbo.htm

[4] dark spyrit, "Win32 Buffer Overflows - Location, Exploitation and Defense", Phrack Magazine 55 (15), May 2000

[5] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools". In Proceedings of the 1994 Conference on Programming Language Design and Implementation (PLDI), pages 196–205, June 1994.

[6] James Larus and Eric Schnarr, "EEL: Machine-independent executable editing". In SIGPLAN Conference on Programming Languages, Design and Implementation, pages 291–300, June 1995.

[7] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, and Brian Bershad. "Instrumentation and optimization of win32/intel executables using Etch". In USENIX Windows NT Workshop, 1997.

[8] "LEEL", www.geocities.com/fasterlu/leel.htm

[9] C. Cifuentes and M. Van Emmerik, "UQBT: Adaptable Binary Translation at Low Cost", IEEE Computer, March 2000.

[10] Crispin Cowan et al., "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks". In Proceedings of the 7th USENIX Security Symposium, pages 63-78, San Antonio, TX, January 1998.

[11] Microsoft compiler extension for buffer overflow defense, http://go.microsoft.com/fwlink/?Linkid=7260

[12] Stackshield, www.angelfire.com/sk/stackshield/

[13] Win32 Disassembler, www.geocities.com/ sangcho

[14] Hiroaki Etoh. "GCC extension for protecting applications from stack-smashing attacks." http://www.trl.ibm.co.jp/projects/security/ssp

[15] "CASH: Checking Array Bound Violation Using Segmentation Hardware", www.ecsl.cs.sunysb.edu/softsecure/project.html

[16] R. Jones and P. Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs", http://www-ala.doc.ic.ac.uk/ phjk/BoundsChecking.html

[17] Intel Architecture Software Developer's Manual: Volume 3: System Programmer's Guide

[18] Bulba and Kil3r. Bypassing StackGuard and StackShield. Phrack, 5(56), May 2000.

[19] Phrack Magazine 55 (8), May 2000: Klog - The frame pointer overwrite

[20] Arash Baratloo, Timothy Tsai, and Navjot Singh, "Transparent run-time defense against stack smashing attacks" In Proceedings of the USENIX Annual Technical Conference, June 2000.

[21] Vladimir Kiriansky, Derek Bruening, Saman Amarasinghe, "Secure Execution Via Program Shepherding", 11th USENIX Security Symposium, August 2002, San Francisco, California.

[22] Benjamin Schwarz, Saumya Debray, Gregory Andrews, "Disassembly of executable code revisited", Working Conference on Reverse Engineering, Oct 2002.

[23] C. Cifuentes, M. Van Emmerik, "Recovery of Jump Table Case Statements from Binary Code", International Workshop on Program Comprehension, May 1999

[24] Galen Hunt and Doug Brubacher, "Detours: Binary Interception of Win32 Functions". Proceedings of the third Usenix NT Symposium, Seattle, July 1999.

[25] Matt Pietrek, "An In-Depth Look into the Win32 Portable Executable File Format", MSDN magazine, Feb 2002

[26] Matt Pietrek, Under the Hood, Microsoft Systems Journal, 11(5), May 1996.