A Comparison of Buffer Overflow Prevention Implementations and Their Weaknesses

Richard Johnson | Peter Silberman

Agenda

- Compiler-Enforced Protection
  - StackGuard
  - StackShield
  - ProPolice
  - Microsoft /GS Compiler Flag
- Kernel-Enforced Protection
  - PaX
  - StackDefender 1 & 2
  - OverflowGuard
- Attack Vector Test Platform

Compiler-Enforced Protection

Compiler-Enforced Approach

- Advantages
  - No system-wide performance impact
  - Intimate knowledge of binary structure
- Disadvantages
  - Requires modification of each protected binary (including shared libraries) and source code must be available
  - Protections must account for each attack vector since execution environment is not protected

Compiler-Enforced Concepts

- Buffer Overflow Prevention is accomplished by protecting control data stored on the stack.
- Re-ordering Stack Variable Storage
- Stack Canaries
  - Random Canary
  - Random XOR Canary
  - Null Canary
  - Terminator Canary

StackGuard

- Pioneered the use of stack canaries.
- Modifications to the function_prologue and function_epilogue generate and validate canaries.
- Canary originally adjacent to return address.
- Latest version protects both return address and frame pointer.
- Canary location is now architecture specific.

StackShield

- Global Ret Stack
  - Return address is placed in the Global Ret Stack whenever a function is called and copied out whenever the function returns.

- Ret Range Check
  - Copies return address to non-writable memory in function_prologue
  - function_epilogue checks against stored return address to detect an overflow.

- Function pointers are also checked to ensure they point to the .text section.

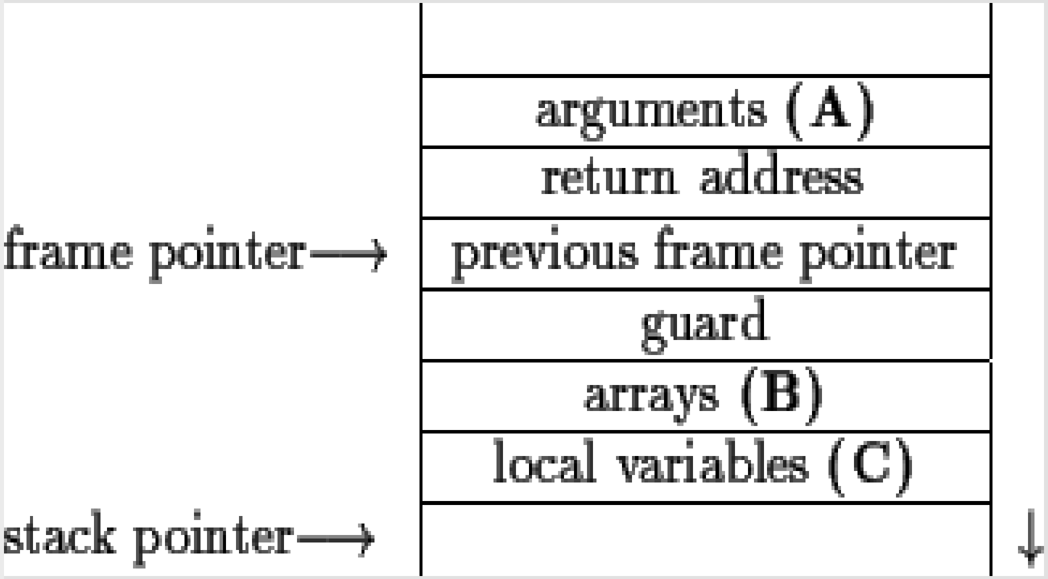
ProPolice SSP

- Implements a safe stack model which rearranges argument locations, return addresses, previous frame pointers and local variables.

- Provides most complete buffer overflow prevention solution of all evaluated compiler-enforced protection software.

ProPolice SSP

- Arrays and local variables are all below the return address.



ProPolice SSP

- Vulnerable code segment (provided by ProPolice docs):

```
void bar( void (*func1)() )
{
        void (*func2)();
        char buf[128];
        .......
        strcpy (buf, getenv ("HOME"));
        (*func1)(); (*func2)();
}
```

- In our example, an overflow in buf could overwrite the function pointers. However, SSP will change this code to….

ProPolice SSP

```
void bar( void (*tmpfunc1)() )
{
        char buf[128];
        void (*func2)();
        void (*func1)(); func1 = tmpfunc1;
        .......
        strcpy (buf, getenv ("HOME"));
        (*func1)(); (*func2)();
}
```

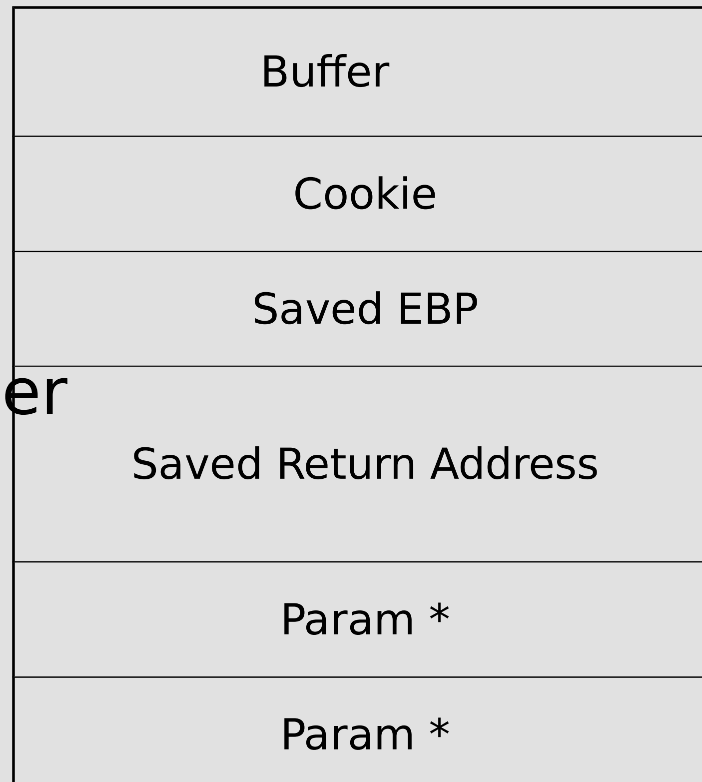
Using the ProPolice safe stack, the passed function pointer is put in a register or local variable by the compiler.

Microsoft Compiler Extension

- Initial release of Microsoft's .NET compiler included buffer overflow protection
- .NET compiler protection is a re-incarnation of Crispin Cowan's StackGuard
- Differences
  - Cookies vs. Canaries
  - Storing in Writable Memory

How the /GS Switch Works

- The GS switch adds a security cookie
- When the cookie check occurs:
  - Original cookie stored in .data section
  - Compared to the cookie on the stack
  - No match security handler called
- Modifications to Exception Handler
  - Can’t point to stack
  - Registered Handler



.NET Protection Bypass

- Exception Handler Bypass
  - Exception handler points to heap
  - Exception handler points to registered handler

- If the attacker has an arbitrary DWORD overwrite
  - Overwrite the saved cookie
  - Overwrite the security handler function pointer

Kernel-Enforced Protection

Kernel-Enforced Approach

- Advantages
  - Does not require source code or modifications to binaries
  - Kernel has control over the MMU

- Disadvantages
  - Architecture/platform dependant
  - Noticeable performance impact on architectures that don’t natively support non-executable features

Kernel-Enforced Concepts

- Buffer Overflow Prevention is accomplished by applying access controls to the MMU and randomizing process memory layout.

- The goal of kernel-enforced buffer overflow protection is to prevent and contain the following:
  - Introduction/execution of arbitrary code
  - Execution of existing code out of original program order
  - Execution of existing code in original program order with arbitrary data

Memory Management Unit Access Control Lists

- Non-executable (NOEXEC) protection is the most commonly used access control for memory.

- A non-executable stack resides on a system where the kernel is enforcing proper "memory semantics."
  - Separation of readable and writable pages
  - All executable memory including the stack, heap and all anonymous mappings must be non-executable.
  - Deny the conversion of executable memory to non-executable memory and vice versa.

Address Space Layout Randomization

- Defeats rudimentary exploit techniques by introducing randomness into the virtual memory layout of a process.

- Binary mapping, dynamic library linking and stack memory regions are all randomized before the process begins executing.

PaX

- PaX Project's kernel patches provide an example of one of the more robust kernel-based protection software currently available.

- PaX offers prevention against unwarranted code execution via memory management access controls and address space randomization.

PaX NOEXEC

- NOEXEC aims to prevent execution of arbitrary code in an existing process’s memory space.

- Three features which ultimately apply access controls on mapped pages of memory:
  - executable semantics are applied to memory pages
  - stack, heap, anonymous memory mappings and any section not marked as executable in an ELF file is non-executable by default.
  - ACLs on mmap() and mprotect() prevent the conversion of the default memory states to an insecure state during execution (MPROTECT).

PaX PAGEEXEC

- Implementation of non-executable memory pages that is derived from the paging logic of IA-32 processors.
- Pages may be marked as “non-present” or “supervisor level access”.
- Page fault handler determines if the page fault occurred on a data access or instruction fetch.
  - Instruction fetch – log and terminate process
  - Data access – unprotect temporarily and continue

PaX SEGMEXEC

- Derived from the IA-32 processor segmentation logic

- Linux runs in protected mode with paging enabled on IA-32 processors, which means that each address translation requires a two step process.
  - LOGICAL <-> LINEAR <-> PHYSICAL

- The 3gb of userland memory space is divided in half:
  - Data Segment: 0x00000000 - 0x5fffffff
  - Code Segment: 0x60000000 – 0xbfffffff

- Page fault is generated if instruction fetches are initiated in the non-executable pages.

PaX MPROTECT

- Prevents the introduction of new executable code to a given task's address space.

- Objective of the access controls is to prevent:
  - Creation of executable anonymous mappings
  - Creation of executable/writable file mappings
  - Making executable/read-only file mapping writable except for performing relocations on an ET_DYN ELF
  - Conversion of non-executable mapping to executable

PaX MPROTECT

- Every memory mapping has permission attributes which are stored in the vm_flags field of the vma structure within the Linux kernel.

- The four attributes which define the permissions of a particular area of mapped memory are:
  - VM_WRITE
  - VM_EXEC
  - VM_MAYWRITE
  - VM_MAYEXEC

PaX MPROTECT

- The Linux kernel requires VM_WRITE enabled if the VM_MAYWRITE attribute is true. Also applies to VM_EXEC.

- PaX must deny WRITE and EXEC permissions on the same page leaving the safe states to be:
  - VM_MAYWRITE
  - VM_MAYEXEC
  - VM_WRITE | VM_MAYWRITE
  - VM_EXEC | VM_MAYEXEC

PaX ASLR

- Address Space Layout Randomization (ASLR) renders exploits which depend on predetermined memory addresses useless by randomizing the layout of the virtual memory address space.

- PaX implementation of ASLR consists of:
  - RANDUSTACK
  - RANDKSTACK
  - RANDMMAP
  - RANDEXEC

PaX RANDUSTACK

- Responsible for randomizing userspace stack.
- Kernel creates program stack upon each execve() system call.
  - Allocate appropriate number of pages
  - Map pages to process's virtual address space
    - Userland stack usually is mapped at 0xbfffffff
- Randomization is added both in the address range of kernel memory to allocate and the address at which the stack is mapped.

PaX RANDKSTACK

- Responsible for randomizing a task's kernel stack
- Each task is assigned two pages of kernel memory to be used during the execution of system calls, interrupts, and exceptions.
- Each system call is protected because the kernel stack pointer will be at the point of initial entry when the kernel returns to userspace

PaX RANDMMAP

- Handles the randomization of all file and anonymous memory mappings.
- Linux usually allocates heap space by beginning at the base of a task's unmapped memory and locating the nearest chunk of unallocated space which is large enough.
- RANDMMAP modifies this functionality in do_mmap() by adding a random delta_mmap value to the base address before searching for free memory.

PaX RANDEXEC

- Responsible for randomizing the location of ET_EXEC ELF binaries.
  - Image must be mapped at normal address with pages set non-executable
  - Image is copied to random location using RANDMMAP logic.
- Page fault handler will handle accesses to both binary images and allow access when proper conditions are met.

NGSEC StackDefender 1.10

- StackDefender implements a unique protection
  - Protection based on ACLs surrounding API calls

- StackDefender files:
  - kernelNG.fer
  - msvcNG.fer
  - ntdNG.fer
  - Proxydll.dll
  - StackDefender.sys

StackDefender.sys

- Hooks ZwCreateFile, ZwOpenFile to detect:
  - kernel32.dll
  - msvcrt.dll
  - ntdll.dll
- Redirect files to:
  - *NG.fer

Understanding System Calls

```
__asm
{
     mov eax, 0x64
     lea edx, [esp+0x04]
     int 0x2e
}
```

- Gateway between User-mode and Kernel-mode
  - KiSystemService
  - `call KeServiceDescriptorTable->ServiceTableBase[function_id]`

Hooking System Calls

```
__asm
{
     cli                                    ; stop interrupts
     mov edx, ds:ZwCreateFile               ; save function pointer
     mov ecx, ds:KeServiceDescriptorTable  ; save KeSDT pointer
     mov ecx, [ecx]                         ; Get base
     mov edx, [edx+1]                       ; Get function number
     mov edx, [ecx+edx*4]                   ; ServiceTableBase
     mov old_func, edx                      ; store old function
     mov edx, [edx+1]
     mov dword ptr [ecx+edx*4], offset function_overwrite
     sti
}
```

NG.fer Files

- Used by StackDefender to add randomness to the systems DLL’s image base.

- Makes a copy of system DLLs
  - Kernel32.dll
  - Ntdll.dll
  - Msvcrt.dll

What is the Export Address Table (EAT)?

- Used to export a function for other processes

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;     // RVA from base of image
    DWORD   AddressOfNames;         // RVA from base of image
    DWORD   AddressOfNameOrdinals;  // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

- To resolve a function export:
  - Obtain the Virtual address of the EAT
  - Walk AddressOfNames, and AddressOfNameOrdinals
  - Index AddressOfFunctions

kernelNG.fer

- Setup KernelNG.fer
  - Modify characteristics of the .reloc section
    - 42000040 (Readable + Discardable + Initialized Data)
    - E2000060 (Executable + Writable + Readable)
  - Copy function stubs
  - Implement Export Address Table Relocation
    - Overwrites function entry point

kernelNG.fer (cont.)

StackDefender overwrites the following function’s EAT entries:

- WinExec
- CreateProcessA
- CreateProcessW
- CreateThread
- CreateRemoteThread
- GetProcAddress
- LoadModule
- LoadLibraryExA
- LoadLibraryExW
- OpenFile
- CreateFileA
- CreateFileW
- _lopen
- _lcreat
- CopyFileA
- CopyFileW
- CopyFileExA
- CopyFileExW
- MoveFileA
- MoveFileExW
- MoveFileWithProgressA
- MoveFileWithProgressW
- DeleteFileA
- LockFile
- GetModuleHandleA
- VirtualProtect
- OpenProcess
- GetModuleHandleW

StackDefender Overflow Detection

- .reloc from kernelng.fer loads proxydll.dll

- Proxydll.dll exports StackDefender()
  - arg1 = esp+0x0C
  - arg2 = where the function was called from
  - arg3 = integer
  - arg4 = stack address of a parameter

- Proxydll overflow detection
  - Alert API Routine
    - Checks API for strings e.g. cmd.exe
  - Calls VirtualQuery() on arg1 and arg2
    - MEMORY_BASIC_INFORMATION->AllocationBase
  - IsBadWritePtr() called on arg2

Defeating StackDefender

- Shellcode that puts itself on the heap and marks the heap read-only

- Shellcode that calls ntdll functions e.g. ZwProtectVirtualMemory
  - Bypasses API hooks

StackDefender 2.00

- Heavily influenced by PaX
- Moved away from API ACL
- Initial Analysis shows:
  - Hooks ZwAllocateVirtualMemory and ZwProtectVirtualMemory
  - Hooks int 0x0e and int 0x2e

Vulnerabilities in StackDefender

- StackDefender 1.10
  - Blue Screen of Death when calling ZwCreateFile / ZwOpenFile with an invalid ObjectAttribute parameter.

- StackDefender 2.00
  - Blue Screen of Death when ZwProtectVirtualMemory is given an invalid BaseAddress

DataSecuritySoftware OverflowGuard 1.4

- OverflowGuard implements PaX page protection

- OverflowGuard hooks Interrupt Descriptor Table entries 0x0e and 0x01.
  - 0x01 -> Debug Exception
  - 0x0e -> Page Fault

- OverflowGuard Files:
  - OverflowGuard.sys

What is the Interrupt Descriptor Table (IDT)?

- Provides array of function pointers as handlers for userland exceptions or events

- Kernel receives interrupt request and dispatches the correct handler

- Interrupt or Exception occurs
  - int 0x03 - breakpoint
  - int 0x0e - invalid memory access

Overwriting IDT

- Use sidt instruction to obtain IDT base
- Load address of interrupt handler
  - IDT base addr + interrupt id * 8
- The Interrupt Gate which OverflowGuard needs to overwrite looks like:

31-16	15	14-13	12-8	7-5	4-0
Offset	P	DPL	0-D-1-1-0	0-0-0	Reserved

Segment Selector	15-0
	Offset

OverflowGuard Buffer Overflow Protection

- OverflowGuard sets memory mappings to read-only
- Writing stack or heap when its in read-only mode
  - Causes page fault
    - Updates Permissions
- Page Fault Handler
  - OverflowGuard converts old EIP to physical address
    - Compares old EIP to fault address
      - Then it was an execution attempt
      - Otherwise it was a data access
        - » Find memory address
        - » Mark it writable/user/dirty
        - » Perform dummy read
        - » Reset memory permissions to supervisor

Defeating OverflowGuard

- Return-into-libc previously demonstrated by ins1der
- Does not protect third party software

Attack Vector Test Platform

Attack Vector Test Platform

- Provides objective test results to determine gaps in buffer overflow prevention software
- Simulates exploitation of various attack vectors
- Original work by John Wilander

Attack Vector Test Platform Results

	PaX	StackGuard	StackShield	ProPolice SSP	Visual Studio .NET	OverflowGuard	StackDefender 1.10	StackDefender 2.0
Stack overflow to target								
Parameter function pointer	+	-	-	+	+	-	+	-
Parameter longjmp buffer	+	-	-	-	N/A	N/A	N/A	N/A
Return address	+	+	+	+	+	-	+	-
Old base pointer	+	+	+	+	N/A	N/A	N/A	N/A
Function pointer	+	-	-	+	+	-	+	-
Longjmp buffer	+	-	-	+	N/A	N/A	N/A	N/A
Heap/BSS overflow to target								
Function pointer	+	-	-	-	N/A	N/A	N/A	N/A
Longjmp buffer	+	-	-	-	N/A	N/A	N/A	N/A
Pointer on stack								
Parameter function pointer	+	-	-	+	+	-	+	-
Parameter longjmp buffer	+	-	-	+	N/A	N/A	N/A	N/A
Return address	+	-	+	+	+	-	+	-
Old base pointer	+	+	+	+	N/A	N/A	N/A	N/A
Function pointer	+	-	-	+	+	-	+	-
Longjmp buffer	+	-	-	+	N/A	N/A	N/A	N/A
Pointer on heap/BSS								
Return address	+	-	+	-	N/A	N/A	N/A	N/A
Old base pointer	+	+	+	+	N/A	N/A	N/A	N/A
Function pointer	+	-	-	-	N/A	N/A	N/A	N/A
Longjmp buffer	+	-	-	-	N/A	N/A	N/A	N/A

Conclusion

- Test results show that there are varying coverage capabilities in the available protection software
- Windows protection has not advanced yet
  - Few compiler options
  - Successful protection of third party applications
- Combination of kernel and compiler-based protection software is currently the best defense.

Thanks

Special thanks go out to:

- Matt Miller for technical insight and research verification
- Lord YuP for conceptual contributions

We'd also like to thank:

- iDEFENSE Labs, Dr Dobbs Journal for lending us articles to read, Dr. John Wilander for initial Testbed, and StackDefender Development team for being affable and helpful throughout the research process.

Questions?