StackGuard: A Historical Perspective

Crispin Cowan, PhD
Senior PM, Windows Core Security
Microsoft
Aleph One Fires The Opening Shot

• “Smashing the Stack for Fun and Profit”
  – Aleph One (AKA Elias Levy), Phrack 49, August 1996

• It is a cook book for how to create exploits for “stack smashing” attacks

• Prior to this paper, buffer overflow attacks were *known*, but not widely exploited
  – “Validate all input parameters” is a security principle going back to the 1960s

• After this paper, attacks became rampant
  – Stack smashing vulns are massively common, easy to discover, and easy to exploit
What is a “Stack Smash”?

• Buffer overflow:
  – Program accepts string input, placing it in a buffer
  – Program fails to correctly check the length of the input
  – Attacker gets to overwrite adjacent state, corrupting it

• Stack Smash:
  – Special case of a buffer overflow that corrupts the activation record
What is a “Stack Smash”? 

• Return address
  – Overflow changes it to point somewhere else

• “Shell Code”
  – Point to exploit code that was encoded as CPU instructions in the attacker’s string
  – That code does `exec("/bin/sh")`
    hence “shell code”
Why Are We So Vulnerable To Something So Trivial?

- Why are we so vulnerable to something so trivial?
  - Because C chose to represent strings as null terminated instead of (base, bound) tuples
  - Because strings grow up and stacks grow down
  - Because we use Von Neumann architectures that store code and data in the same memory

- But these things are hard to change ... mostly
Non-Executable Memory

• Try to move away from Von Neumann architecture by making key regions of memory be non-executable

• Problem: x86 memory architecture does not distinguish between “readable” and “executable” per page
  – Only memory segments support this distinction
  – Most other CPU memory systems support non-executable pages, but they also mostly don’t matter 😊
Non-Executable Stack, 1997

• “Solar Designer” introduces the Linux non-executable stack patch
  – Fun with x86 segmentation registers maps the stack differently from the heap and static data
  – Results in a non-executable stack
• Effective against naïve Stack Smash attacks
• Bypassable:
  – Inject your shell code into the heap (still executable)
  – Point return address at your shell code in the heap
StackGuard, 1998

• Compile in integrity checks for activation records
  – Insert a “canary word” (after the **Welsh miner’s canary**)

• If the canary word is damaged, then your stack is corrupted
  – Instead of jumping to attacker code, abort the program
  – Log the intrusion attempt
StackGuard Prototype

• Written in a few days by one intern
• Less than 100 lines of code patch to GCC
  – Helped a lot that the GCC function preamble and function post amble code generator routines were nicely isolated
• First canary was hardcoded 0xDEADBEEF
  – Easily spoofable, but worked for proof of concept
Canary Spoof Resistance

• The random canary:
  – Pull a random integer from the OS /dev/random at process startup time
  – Simple in concept, but in practice it is very painful to make reading from /dev/random work while still inside crt0.o
  – Made it work, but motivated us to seek something simpler

• “Terminator” canary:
  – CR, LF, 00, -1: the symbols that terminate various string library functions
  – Rationale: will cause all the standard string mashers to terminate while trying to write the canary → cannot spoof the canary and successfully write beyond it
  – Still vulnerable to attacks against poorly used memcpy() code, but buffer overflows thought to be rare
XOR Random Canary

• 1999, “Emsi” creates the frame pointer attack
  – Frame pointer stored below the canary → corruptible
  – Change FP to point to a fake activation record constructed on the heap
  – Function return code will believe FP, interpret the fake activation record, and jump to shell code
  – Bypasses both Terminator and Random Canaries

• XOR Random Canary
  – XOR the correct return address with the random canary
  – Integrity check must match both the random number, and the correct return address
Other Stack Smashing Defenses

• StackShield:
  – Copied valid return addresses to safe memory, check them on function return
  – Implemented as a modified assembler → requires hacking your makefiles

• Libsafe: armored variants of the “big 7” standard string library functions
  – Library code does a plausability check on the parameters; ensure that they are not pointing back up the stack at an activation record
  – Advantage: no recompile necessary
  – Disadvantage: no protection for hand-coded string handling, or anything other than the big-7
Other Stack Smashing Defenses

• StackGhost: uses SPARC CPU hardware to get OS in the loop to armor the stack
• Hardware: numerous papers proposing “slightly” modified CPU hardware to protect against stack smashing
  – Typically protection about as good as StackGuard
  – Advantage: don’t have to re-compile code
  – Disadvantage: do have to re-compile code to run on non-existent hardware, which tends to limit adoption 😊
StackGuard Derivatives: ProPolice

• IBM Research Japan
  – Also a modified GCC
  – Copied StackGuard defense exactly, and acknowledged it
  – Enhanced with variable sorting: sort buffers (arrays) up to the top of local variables, so that they cannot overflow other important values

• Used a different code generator technique
  – More compatible with the newer code generator architecture in GCC 2 and GCC 3
  – Ultimately ProPolice is what is adopted into GCC and became the \texttt{-fstack	extunderscore protector} feature
StackGuard, uh ...

Concurrent Innovation 😊

• Microsoft Visual Studio: /gs
  – Uses exactly the StackGuard defense
  – Introduced in 2003; people who were there say that it was independently innovated
  – Object lesson: **patent your stuff, even if you intend to GPL it!**

• Even though introduced 5 years after StackGuard, Microsoft beat the Linux/FOSS community into mainstream adoption by several years
All the World Is Not A Stack

• As stack protection matured, attackers do what they always do: move to the next soft target
  – Heap overflows
  – Pointer corruption
  – Printf format string vulnerabilities
  – Integer “underflows”
  – ...
Brute Force Defense: Buffer Bounds Checking

• Jones&Kelly built a GCC that had **full** array bounds checking
  – Associate a data structure with every buffer and check every read and write against the buffer’s legitimate size
  – Absolutely memory safe
  – Costly: between 3X and 30X slowdown
Fun With Memory Defense: DEP and ASLR

- DEP: Data Execution Protection
- ASLR: Address Space Layout Randomization
- Microsoft introduced in XPSP2
- Linux introduced bits and pieces in various places:
  - PAX Project also had NX (Like DEP) and ASLR
  - Red Hat ExecShield
DEP and ASLR Are Critically Interdependent

- ASLR only: not enough bits of randomization
  - Attacker can inject their code surrounded by a “NOP sled”; long sequence of NOPs followed by shell code
  - Only have to jump to somewhere in the NOP sled to succeed
  - Add DEP: cannot inject code into data areas

- DEP only: there is lots of code in memory already that can do the attacker’s job
  - Originally called the “return into LibC” attack; the attacker changes the return pointer to point to some code in LibC that will run exec(“/bin/sh”)
  - Add ASLR: becomes hard for the attacker to hit that delicate target, because they cannot surround it with a NOP sled
PointGuard

- Cowan et al, USENIX Security 2003
- Hashed pointers; the *dual* of ASLR
- Pointers in memory: can be corrupted via overflow
- Pointers in registers: not overflowable
- PointGuard:
  - Store pointers *encrypted* in memory
  - To dereference a pointer, decrypt it as you load it into a register
Normal Pointer Dereference

1. Fetch Pointer Value
2. Access data referenced by pointer
Normal Pointer Dereference Under Attack

1. Fetch Pointer Value
2. Access attacker’s data referenced by corrupted pointer

CPU

Memory

Corrupted Pointer
0x1234
0x1340

Data
Malicious Data

0x1234
0x1340

CPU

1. Fetch Pointer Value
2. Access attacker’s data referenced by corrupted pointer
**PointGuard Pointer Dereference**

1. Fetch Pointer Value
2. Access data referenced by pointer

Diagram:
- CPU
- Pointer Decryption
- Encrypted Pointer 0x7239
- Data
- Memory

Values:
- 0x1234
- 0x7239
- 0x1234
PointGuard Pointer Dereference Under Attack

1. Fetch Pointer Value
2. Access random data referenced by decryption of corrupted pointer
3. Segfault & Crash

Memory

CPU

0x9786

Pointer Decryption

Corrupted Pointer
0x7239
0x1340

Data
Malicious Data
0x1234 0x1340
PointGuard Problems

• PointGuard had excellent performance

• Compatibility not so good: each PG process had its own random cookie
  – Interfacing PG code with non-PG libraries
  – Interfacing PG code with the kernel
  – Bizzare casting: real code declares a union of two structs
    • One variant has a field that is a void *
    • Other variant has that same field as an int
    • The code expects a NULL pointer to show up as an int value == 0, which is **not true** under PG

• PointGuard abandoned due to insurmountable compat issues
  – ASLR and DEP can handle this
Buffer Overflows Today

- **Heap Spray**: fill heap with many many copies of the NOP sled/shell code, to defeat ASLR defenses
- **JIT Spray**: Heap Spray applied to the storage for JIT code, so as to bypass ASLR *and* DEP
- Wise but useless: whatever code shared an address space with the JIT buffer should have been written in a type safe language
- Research opportunity: find a way to defend against JIT Spray that allows people to share JIT address space with crap code 😊
Conclusion

• This is going to keep happening until people adopt type safe languages: Java, C#, Python, Ruby …
  – **Not** C++: it has the safety of C, and the performance of SmallTalk 😊

• But go ahead, keep writing code in insecure languages
  – It is job security for us security nerds

• Questions?
  – Crispin@microsoft.com