Buffer Overflow Attack: Defense Techniques

Countermeasures

- We can take countermeasures at different points in time
  - before we even begin programming
  - during development
  - when testing
  - when executing code
- to prevent, to detect – at (pre)compile time or at runtime -, and to migitate problems with buffer overflows

Preventing Buffer Overflow Attacks

- Non-executable stack
- Static source code analysis.
- Run time checking: StackGuard, Libsafe, SafeC, (Purify).
- Randomization.
- Type safe languages (Java, ML).
- Detection deviation of program behavior
- Sandboxing
- Access control …

Prevention

- Don’t use C or C++ (use type-safe language)
  - Legacy code
  - Practical?
- Better programmer awareness & training
  - Building Secure Software, J. Viega & G. McGraw, 2002
  - Writing Secure Code, M. Howard & D. LeBlanc, 2002
  - 19 deadly sins of software security, M. Howard, D LeBlanc & J. Viega, 2005
  - Secure programming for Linux and UNIX HOWTO, D. Wheeler, www.dwheeler.com/secure-programs
  - Secure C coding, T. Sirainen www.irccrew.org/~cras/security/c-guide.html

Dangerous C system calls

source: Building secure software, J. Viega & G. McGraw, 2002

Extreme risk
- gets

High risk
- strcpy
- strcat
- sprintf
- scanf
- sscanf
- fscanf
- vfscanf
- vsscanf

High risk (cntd)
- streadd
- strecpy
- strtrns
- realpath
- syslog
- getenv
- getopt
- getopt_long
- getpass

Moderate risk
- getchar
- fgetc
- getc
- read
- bcopy

Low risk
- fgets
- memcpy
- snprintf
- strccpy
- strcadd
- strncpy
- strncat
- vsnprintf

Secure Coding

- **Avoid risky programming constructs**
  - **Use fgets instead of gets**
  - **Use strn* APIs instead of str* APIs**
  - **Use snprintf instead of sprintf and vsprintf**
  - **scanf & printf: use format strings**
- **Never assume anything about inputs**
  - **Negative value, big value**
  - **Very long strings**

Prevention – use better string libraries

- there is a choice between using statically vs dynamically allocated buffers
  - static approach easy to get wrong, and chopping user input may still have unwanted effects
  - dynamic approach susceptible to out-of-memory errors, and need for failing safely

Better string libraries

- **libsafe.h** provides safer, modified versions of eg strcpy
- **strlcpy**(dst,src,size) and **strlcat**(dst,src,size**)** with size the size of dst, not the maximum length copied.
  - Used in OpenBSD
- **glib.h** provides Gstring type for dynamically growing null-terminated strings in C
  - but failure to allocate will result in crash that cannot be intercepted, which may not be acceptable
- **Strsafe.h** by Microsoft guarantees null-termination and always takes destination size as argument
- **C++ string class**
  - data() and c-str()return low level C strings, ie char*, with result of data()is not always null-terminated on all platforms...

Dynamic countermeasures

- Protection by kernel
  - Non-executable stack memory (NOEXEC)
    - prevents attacker executing her code
  - Address space layout randomisation (ASLR)
    - generally makes attacker's life harder
      - E.g., harder to get return address place and injected code address
- Protection inserted by the compiler
  - to prevent or detect malicious changes to the stack
- Neither prevents against heap overflows

Bugs to Detect in Source Code Analysis

- Some examples

- Crash Causing Defects
- Null pointer dereference
- Use after free
- Double free
- Array indexing errors
- Mismatched array new/delete
- Potential stack overrun
- Potential heap overrun
- Return pointers to local variables
- Logically inconsistent code

- Uninitialized variables
- Invalid use of negative values
- Passing large parameters by value
- Underallocations of dynamic data
- Memory leaks
- File handle leaks
- Network resource leaks
- Unused values
- Unhandled return codes
- Use of invalid iterators

Marking stack as non-execute

- Basic stack exploit can be prevented by marking stack segment as non-executable or randomizing stack location.
  - Then injected code on stack cannot run
  - Code patches exist for Linux and Solaris
    - E.g., our olympus.eecs.ucf.edu has patched for stack radnomization
- Problems:
  - Does not block more general overflow exploits:
    - Overflow on heap, overflow func pointer
  - Does not defend against `return-to-libc' exploit.
  - Some apps need executable stack (e.g. LISP interpreters).

Randomization Techniques

- **For successful exploit, the attacker needs to know where to jump to, i.e.,**
  - **Stack layout for stack smashing attacks**
  - **Heap layout for code injection in heap**
  - **Shared library entry points for exploits using shared library**
- ***R*****andom*****i*****zation Techniques for Software** ***S*****ecurity**
  - **Randomize system internal details**
    - **Memory layout**
    - **Internal interfaces**
  - **Improve software system security**
    - **Reduce attacker knowledge of system detail to thwart exploit**
    - **Level of indirection as access control**

Randomize Memory Layout (I)

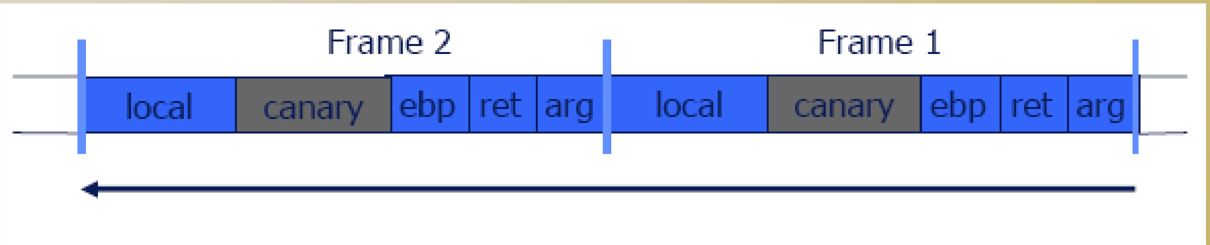
- **Randomize stack starting point**
  - **Modify execve() system call in Linux kernel**
  - **Similar techniques apply to randomize heap starting point**
- **Randomize heap starting point**
- **Randomize variable layout**

Randomize Memory Layout (II)

- **Handle a variety of memory safety vulnerabilities**
  - **Buffer overruns**
  - **Format string vulnerabilities**
  - **Integer overflow**
  - **Double free**
- **Simple & Efficient**
  - **Extremely low performance overhead**
- **Problems**
  - **Attacks can still happen**
    - **Overwrite data**
    - **May crash the program**
  - **Attacks may learn the randomization secret**
    - **Format string attacks**

Dynamic countermeasure: stackGuard

- **Solution: StackGuard**
  - **Run time tests for stack integrity.**
  - **Embed “canaries” in stack frames and verify their integrity prior to function return.**



Canary Types

- **Random canary:**
  - **Choose random string at program startup.**
  - **Insert canary string into every stack frame.**
  - **Verify canary before returning from function.**
  - **To corrupt random canary, attacker must learn the random string.**

Canary Types

- Additional countermeasures:
  - use a random value for the canary
  - XOR this random value with the return address
  - include string termination characters in the canary value (why?)

- StackGuard implemented as a GCC patch
  - Program must be recompiled
- Low performance effects: 8% for Apache
- Problem
  - Only protect stack activation record (return address, saved ebp value)

Purify

- A tool that developers and testers use to find memory leaks and access errors.
- Detects the following at the point of occurrence:
  - reads or writes to freed memory.
  - reads or writes beyond an array boundary.
  - reads from uninitialized memory.

Purify - Catching Array Bounds Violations

- **To catch array bounds violations, Purify allocates a small "red-zone" at the beginning and end of each block returned by malloc.**
- **The bytes in the red-zone → recorded as unallocated.**
- **If a program accesses these bytes, Purify signals an array bounds error.**
- **Problem:**
  - **Does not check things on the stack**
  - **Extremely expensive**

Further improvements

- PointGuard
  - also protects other data values, eg function pointers, with canaries
    - Higher performance impact than stackGuard
- ProPolice's Stack Smashing Protection (SSP) by IBM
  - also re-orders stack elements to reduce potential for trouble
- Stackshield has a special stack for return addresses, and can disallow function pointers to the data segment

Dynamic countermeasures

- libsafe library prevents buffer overruns beyond current stack frame in the dangerous functions it redefines
  - Dynamically loaded library.
  - Intercepts calls to strcpy (dest, src)
    - Validates sufficient space in current stack frame:

      |frame-pointer – dest| > strlen(src)
    - If so, does strcpy.
      Otherwise, terminates application.

Dynamic countermeasures

- libverify enhancement of libsafe keeps copies of the stack return address on the heap, and checks if these match

- None of these protections are perfect!
  - even if attacks to return addresses are caught, integrity of other data other than the stack can still be abused
  - clever attacks may leave canaries intact
  - where do you store the "master" canary value
    - a cleverer attack could change it
  - none of this protects against heap overflows
    - eg buffer overflow within a struct...
  - New proposed non-control attack

Summary

- Buffer overflows are the top security vulnerability
- Any C(++) code acting on untrusted input is at risk
- Getting rid of buffer overflow weaknesses in C(++) code is hard (and may prove to be impossible)
  - Ongoing arms race between countermeasures and ever more clever attacks.
  - Attacks are not only getting cleverer, using them is getting easier