Preventing Buffer Overflows Without Programming

- Idea: make the heap and stack non-executable
  - Because many buffer overflow attacks aim at executing code in the data that overflowed the buffer
- Does not prevent "return into libc" overflow attacks
  - Because the return address of the function on the stack points to a standard "C" function (e.g., "system"), this attack does not execute code on the stack
- e.g., ExecShield for Fedora Linux (used to be RedHat Linux)

Canaries on a Stack (Crispin Cowan)

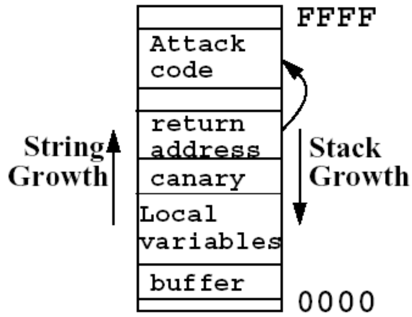
- Add a few bytes containing special values between variables on the stack and the return address.
- Before the function returns, check that the values are intact.
  - If not, there has been a buffer overflow!
    - Terminate program
- If the goal was a Denial-of-Service, then it still happens, but at least the machine is not compromised
- If the canary can be read by an attacker, then a buffer overflow exploit can be made to rewrite it

StackGuard - detect

Add Canary Word next to return address

- Observation (true only for buffer o.f.)
  - Return address is unaltered IFF canary word is unaltered (?)

- Guessing the Canary ?
  - Randomize



StackGuard - detect

- When compiling the function, it adds prologue and epilogue
  - Before execution of function, push word canary into canary vector
    - in addition to the stack
  - After execution, before returning from function check whether canary is intact
  - Function returns ONLY if canary is intact

4

StackGuard – Prevent

- While function is active, make the return address read-only
  - attacker cannot change the return address
  - any attempt will be detected
  - Use a library called MemGuard
- mark virtual memory pages as read-only and trap every write
  - legitimate writes to stack causes trap
  - Performance penalty

5

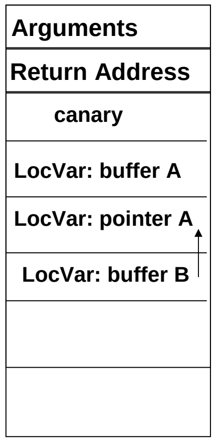
Canary Implementations

- StackGuard
- Stack-Smashing Protector (SSP)
  - gcc modification
  - Used in OpenBSD
  - http://www.trl.ibm.com/projects/security/ssp/
- Windows: /GS option for Visual C++ .NET
- These can be useful when testing too!

6

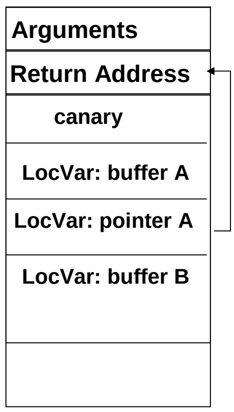
StackGuard Bypass

- Guarding a stack is not the answer, as B.O. is not a stack problem but a pointer problem (controlling a pointer –the instruction pointer in this case-)
- Consider a function with several local variables, some of which are pointers: if we overflow B, we can overwrite pointer A. If this is a function pointer, it will be called, then pointing to our code



StackGuard Bypass (cont.)

- The return address can be overwritten without touching the canary value (trampolining)
- Another possibility is to modify pointer A to point to a structure that holds function pointers, modifying an address there; point one of these back to buffer. If function gets called and buffer still around, control achieved.



Arithmetic Issues:

- In mathematics, integers form an infinite set, but in systems they are binary strings of fixed length (precision), so a finite set. Familiar rules of arithmetic do not apply.
- In unsigned 8-bit integer arithmetic
  1. 255+1= 0,
  2. 16 X 17=16 and
  3. 0-1=255
- In particular, a negative value (as in 3.) can be interpreted as a 'large' positive one

9

Example (using 1.)

Consider the following code snippet that copies two character strings into a buffer and checks the combined length so they fit

```
char buf [128]
combine(char *s1, size_t len1, char *s2,size_t
   len2) {
      if (len1+len2+1 <= sizeof(buf)) {
      strncpy(buf, s1, len1);
      strncat(buf, s2, len2);   }
   }
```

The system could be attacked by constructing s1 so that `len1<= sizeof(buf)` and set `len2=0xFFFFFFFF`

(as unsigned integer, it corresponds to 4294967295)

Now, since `len1+0xFFFFFFFF+1 = len1 <=sizeof(buf))`

The `strncat` is executed and the buffer overrun.

10

Example (using 3.)

Consider the following code snippet

```
int main(int argc, char* argv[])
  {   char _t[10]
      char p[]="xxxxxxx";
      char k[]="zzzz";
      strncpy(_t, p, sizeof(_t));
      strncat(_t, k, sizeof(_t) – strlen(_t)-1);
      return 0;
  }
```

After execution, the resulting string in `_t` is `xxxxxxxzz;`

Now if we supply 10 chars in p (`xxxxxxxxxx`), then `sizeof(_t)` and `strlen(_t)` are equal and the third argument is `-1`.

Since `strncat` expects unsigned as third argument, it is interpreted as `0xFFFFFFFF` and therefore the `strcat` is unbounded and the buffer overrun again.

11

Important Lesson

- Declare all integers as unsigned integers, unless negative ones are really needed. While measuring size of objects, negative ones are not needed. If compiler flags signed-unsigned mismatch, check if both representations are needed; if so, care needed to the checks implemented.
- Most arithmetic bugs are caused by type mismatch

12

Buffer Overflow in Java?

- Not really, since Java has a type-safe memory model, and 'falling off' the end of an object is not possible.
- Exploits against Java-based systems are typically language-based (type confusion) attacks and trust exploits (code signing errors)
- Problem overflow typically occur in supporting code external to the JVM: use, by Java-based services, of components and services written in weakly typed languages like C and C++
- Java supports loading of DLLs and code libraries, so that exported functions can be used directly

example

```
Public class MyJavaPacketEngine extends Thread
{
   public MyJavaPacketEngine ()
        {
                }
   static
   {
   System.loadLibrary(''packet_driver32'');
        }
}
```

Now calls can be made directly to the DLL.
For example

```
wsprintf(lpAdapter->SymbolicLink, TEXT(''\\\\.\\%s%s''),
   DOSNAMEPREFIX, p_AdapterName);
```

Assigns the binding string to an unterminated string buffer