1 Overview

In the last lecture we introduced the buffer overflow attack, by which attackers can overflow the buffer on the stack to execute an arbitrary program. To prevent this attack, one method is to have a non-executable stack, i.e. DEP (Data Execution Protection). Such a stack has the $W \land X$ feature. That is, it is either writable or executable, but not both at the same time.

In this lecture we will further discuss some other buffer overflow attacks and their countermeasures, and then we will start looking at a new topic, cryptology.

2 More on Buffer Overflows

Is the non-executable stack the final solution to prevent buffer overflow attack? Unfortunately not.

2.1 Return-to-libc Attack

**Attack:** Attackers still have a way to work around DEP. Instead of injecting code onto the stack, an attacker can modify the return address on the stack to return to the memory location of some pre-loaded code---specifically the C library functions---to do what he wants, since he can still put his parameters onto the stack frame by overflowing the buffer. This is know as a return-to-libc attack. It was christened as such due to the libc shared library being the source of many powerful system calls.

**Prevention:** This method of attack was severely weakened (and waned in popularity) over time, the foremost reason being the migration of the x86 assembly family to 64-bit architecture. In this mode, the first parameter of a function is loaded into a registers instead of onto the stack. This makes setting up a library call with the desired arguments exceedingly more difficult.

2.2 Return-Oriented Programming

**Attack:** In response to the stumbling block of having the first parameter of a function call placed into a register, what can an attacker do? Use a function with no arguments? No, this is not a suitable workaround because few of these functions do anything powerful enough to be useful.

Ultimately, one exploit technique did rise to the challenge. Let it be described as follows. Find pieces of existing functions that pop values from the stack into registers. Use the RET command (which returns you back to calling frame address) to move between such code pieces.
Thus you find a sequence of “gadgets” \([A_1, A_2, A_3]\) to maneuver between in order to amalgamate one meta-program with the desired exploit. The described technique is called **return-oriented programming**. One can compose a surprisingly high variety of programs in this fashion. In fact, this is Turing complete because x86 is so dense.

One way to defend against both return-to-libc and malicious return-oriented programming is **ASLR** (Address space layout randomization). ASLR effectively jumbles the locations of key areas of the program inside its process address space, thus making it nearly impossible to know the exact address locations of gadgets an attacker can use. (However, there can be limits to the entropy of a randomization space, and there are ways of leaking information about memory layout through format string vulnerabilities.)

### 2.3 Beyond DEP

What ways are there to circumvent stack-mashing buffer overflows in general? Dedicate a register to storing return addresses? No, it is not practical to expect to do so without changing the hardware. Use a parity bit? No. Similarly, there is no clear place to put it without changing hardware. Keep a shadow copy of the return address? Yes, but the details of how this is implemented are important.

How are we going to store this shadow copy? In a global variable? All subsequent functions will have to be aware of it. In the heap? No, that would be difficult to manage. In the stack? Yes, but it cannot simply be the address. We need to protect it. Thus, our solution is to put a “canary” on the stack below the return address as certificate to verify it has not been modified. Assume this canary is universally known and random (i.e., it is calculated at run time). This idea is called a **stack guard**.

Furthermore, who places the canary, the callee or caller? The callee, because the caller has no place to put the canary. Who checks? The callee. Otherwise, it may be too late to take action once a buffer overflow as been detected. We enforce and implement this idea through the compiler. In doing so, we require no changes on the part of user-space software. It does require re-compilation once change has been implemented, however.

### 3 Cryptology

**Terminology:**

- **Cryptography** -- Secret writing.
- **Cryptanalysis** -- Breaking crypto systems.

#### 3.1 History

Three eras of cryptography:

1. **Manual**
Example: Caesar Cipher is a crypt method where each alphabet is replaced by its next nth letter, with n being a fixed number. It is a type of substitution cipher. Since there are only 25 ways to encrypt a plaintext, Brute force could be an effective attack.

Example: Dancing Man is also a type of substitution cipher, where a crypt method where the alphabets are replaced by figures of dancing men. It is susceptible to Frequency Analysis attack.

2. Mechanical

Example: A notable example is the Enigma, which is a type of electro-mechanical rotor cipher machine used during the World War II. However, it was broken since an operator sent the same encrypted message twice, which violated the crypto system’s design.

Example: A modern incarnation of the Enigma is SSL (Secure Sockets Layer), which is often used with the HTTP protocol to ensure secure communication between server and client. As the connection is initiated, a random key is generated and is used to encrypt all the messages before sending. However, there is a similar weakness -- the same message cannot be encrypted by the same key twice. Hence each SSL connection results in a new key.

Some old banking websites only work with Internet Explorer (IE), which is an unsecure web browser. Some people then use a virtual machine (VM) to use IE. VM is a pretty good way to isolate security threats from running programs. It also has a feature to take a snapshot of the guest machine and restore it later. A security risk could rise if a person browses a website with SSL, takes a snapshot in the middle, and comes back to it later; the browser could use the same key and resend some messages.

3. Math & computer e.g. DES (Data Encryption Standard).

3.2 General View and Setup of Cryptography

Players in cryptography:

- Alice & Bob -- Communicators
- Eve -- Eavesdropper
- Mallory -- Active attacker

A crypto system contains:

- E, D -- Encryption/Decryption Algorithms
- K, K$^{-1}$ -- Encryption/Decryption Keys
- M, C -- Plaintext/ciphertext
**Thread Model** Capabilities of attacker. This is only pertaining to crypto system and not the communication of Alice and Bob. The attacker always has the algorithm due to open design principle.

- Known plaintext attack
- Chosen plaintext attack
- Known cyphertext attack
- Chosen cyphertext attack

**Goals of attacker:**
- Recover the keys (more powerful)
- Recover the plain text (less powerful)
- Recover partial information about plaintext

**One-time pad** The key must be secret, must be as long as the message, and can’t be reused.

\[
E_k(m) = m \oplus k
\]
\[
D_k(c) = c \oplus k
\]

**Perfect secrecy**

\[
\forall m_1, m_2, \text{ and } c, P_k[E_k(m_1) = C] = P_k[E_k(m_2) = C]
\]

This means that seeing the cipher text gives the attacker no advantage whatsoever. The best thing they could do is still random guessing. Even if attacker has some prior knowledge of probability of each encryption, there is still no advantage.

So why don’t we all just use this? Well, how would you send the keys? The keys have to be as long as the plaintext, after all. This is infeasible but ideal. When we hone in on more practical methods, we use this as a reference.