Embedded System Design
Buffer Overflow Tutorial

Max Thrun

November 26, 2012

Problem 1

- **In the tutorial we have seen a popular implementation of a 'stack smashing attack' called 'code injection attack' which made use of the NOP Sled technique to execute code on the stack in order to make the processor call/ jump to 'critical_function'. Can the NOP Sled technique be applied on the hypothetical application shown in Figure 1? Why or why not?**

  The application shown in Figure 1 is susceptible to the NOP sled technique. The array 'name' is only allocated to be 4 bytes but the strcpy function does not take this into account. It will keep writing bytes, starting at name[0], into memory until it reaches a null byte. The attacker could pass a string that is long enough to overwrite the return address on the stack, replacing it with a location somewhere in the NOPs. The program execution would then execute all the NOPs down to where the actual exploit code begins.

- **How would you make the processor execute (possibly in an infinite loop) 'critical_function' by buffer overflowing this application?**

  Like stated above, we would pass an input string to 'process_input' large enough to overwrite the return address. We would have the execution jump to somewhere in the beginning of our buffer which will be filled with NOPs. The execution will continue down the NOP sled until it reaches our actual exploit code. The exploit code will simply jump to 'critical_function' at 0x008002EC. If you want to continue calling critical function you can add a jump at the end of your exploit code to jump back a few instructions and call the critical function again. We can modify the exploit string given in the tutorial to jump to 'critical_function' at 0x008002EC:

```
\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a
\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0
\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a
\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29
\x34\x20\x40\x01\x04\x83\x40\x29\x3a\xe8\x3e\x28\x10\xab\x81\x00
```

Problem 2

- **What (minimum) changes would you suggest in the hypothetical application shown in Figure 2 to be secure and prevent buffer overflows? Assume that the arguments to the program (EmployeeName and EmployeeID) can be at max of MAX_INPUT_SIZE length, including the null byte.**

  One simple solution would be to use the 'strlen()' function to ensure the length of EmployeeName and EmployeeID are ¡ MAX_INPUT_SIZE. The new function could look like:

```
void PrintRecord(char *EmployeeName, char *EmployeeID)
{
    char local_record[128];

    if (strlen(EmployeeName) > MAX_INPUT_SIZE ||
        strlen(EmployeeID) > MAX_INPUT_SIZE )
    {
        printf("Employee name and ID must be less than %d chars\n", MAX_INPUT_SIZE);
        return;
    }
    strcpy(local_record, EmployeeName);
    strcat(local_record, ":");
    strcat(local_record, EmployeeID);
    printf("The new record to be saved is %s", local_record);

    return;
}
```

Problem 3

- **'Code injection attack' executes code using NOP Sled technique mostly to spawn a shell by injecting code, for example with the help of network packets. Explain how the Intrusion Detection System (IDS) can be employed to prevent spawning a new shell. Also explain how an attacker can try to avoid getting caught by IDS.**

  An Intrusion Detection System could monitor system calls and look for suspicious calls such as exec(), which may be used to spawn a shell. It could then kill the child process. If the IDS is implemented at the kernel level it could even just prevent the child process from spawning to begin with. As an attacker, a possible way around this would be to not spawn a shell already on the system but to create and send your own as part of the exploit payload. Upon executing the payload the process essentially *becomes* shell with no need to launch an external one. A super advanced exploit may even execute the shell code in a separate thread to allow the original application to continue running.

Problem 4

- **What a stack overflow is and its relevance in embedded system design.**

  A stack overflow is when the execution stack grows beyond the fixed memory that is reserved for it. For example if you have a recursive function that repeatedly adds return addresses on to the stack eventually you will run out of space and your stack will overflow. This is relevant to embedded systems because most embedded systems will be working with a fixed stack size. This requires the programmers to be careful when dealing with deep function calls.

- **How it is different from stack buffer overflow.**

  A buffer overflow is any case where a program writes beyond the memory allocated to it. For example if you allocate an array with length 4 but then strcpy a string a length 5 into it you've caused a buffer overflow. As shown in the above questions, a buffer overflow can be used to corrupt the stack and enable an attacker to run arbitrary code.

Problem 5

- **Explain various steps involved in the embedded system design process. Explain what security measures you would incorporate in those design steps to design a secure system.**

  The embedded system design process can be broken down into the following

  1. Requirements
  2. Specification
  3. System architecture development
  4. Integration and test

  Of these steps the most important to pay careful attention to security would be system development and testing. Most exploits are a result of simple programming errors, such as not checking the length of the string before doing a strcpy. It is unlikely that theses types of issues would be brough up in any kinda of customer requirements or specification discussion and so it is purely up to the developer to implement good coding techniques. Testing is equally as important to security and testing should not only cover the 'working' conditions such as entering names like 'Jane' and 'Bob' but also extreme conditions such as names like 'AAAAAAAAAAAAAAAAA'. Part of the testing strategy should be to break the program in an attempted to tease out corner cases and unaccounted conditions.

Problem 6

- **Explain the Address Space Layout Randomization (ASLR) security technique and its use. Do you think it's a 100% foolproof method to prevent buffer overflow attacks? Why or why not?**

  The address space layout randomization technique involves randomly arranging the positions of things such as the base address of executable, the stack, and loaded libraries, in the processes address space. This makes it difficult to predict the target address for certain attacks. For example if you are trying to execute shell code which you injected on the stack you must first *find* the stack. The ASLR technique relies on the hope that the attacker will not be able to guess or calculate the position of key structures. I do not think this method is 100% foolproof because there is still a chance that the attacker could guess the location. There might also be a way to calculate and predict the locations if you have access to the ASLR algorithm.