Issue : 2

CSI Connect

A tri annual newsletter of
CSI Student Branch
Aishwarya Institute of Management & IT
Udaipur

May 2012 - August 2012

How to Write EFFECTIVE RESEARCH PAPER

On 18th July 2012, a Workshop on "How to write effective research paper" was organized by CSI Student Branch for the MCA Students of Aishwarya Institute of Management and IT. The expert was Mr. Kapil Shrimal (Asst. Professor, AIM & IT) to lay a foundation towards basic understanding in students. The workshop started with the discussion on the concept of research papers. The expert emphasized on the relevance and need of writing research papers among students. The session covered: meaning of research paper, types of research paper, data collection, research methodology, layout of research paper, with examples. Mr. Kapil Shrimal discussed the techniques of writing a research paper, with different sources focusing on primary and secondary data collection. Importance of literature review was also included. Students were inspired and encouraged to participate in the area of Research, in their field of interest, by Dr. Archana Golwalkar (Director AIM & IT).

Professional problem solving with programming languages with special reference to 'C'

Technical talk on Professional problem solving with programming languages with special reference to "C" was organized for MCA students. The expert speaker Mr. Gaurav Vishwakarma (Director, Xavoc International, Udaipur), an IT Professional working, in the Industry, since last 14 Years in IT, Media and Branding Fields. He is considered as one of best logical analyst. Mr Vishwakarma made the students aware of the skills required in the industry. Discussing on the vital aspects of 'C', the expert stressed on the need of strong foundation in programming and outlined the best preparation for project development.

Patron

Dr. Seema Singh
Chairperson & Managing Director
Aishwarya Education Society

Editor

Dr. Archana Golwalkar
(Director, AIM & IT)

Student Members

Aamir Sanwari
(MCA)

Meenakshi Rao
(MCA)

Ritesh Chouhan
(MCA)

Suman
(MCA)

IMPORTANT LINKS

www.aishwaryacollege.org
www.rtu.ac.in
www.csi-india.org
www.ekalavya.it.iitb.ac.in
www.nbrc.ac.in/library/free_ebooks.htm
www.delnet.nic.in

WRITING Secure Code

Writing secure code is a big deal. There are a lot of viruses in the world, and a lot of them rely on exploits in poorly coded programs. Sometimes the solution is just to use a safer language-Java, for instance - that typically runs code in a protected environment (for instance, the Java Virtual Machine). But this isn't always the best option -- if you need top performance, for example, or if you're working on legacy code written in C or C++. And you need to be aware of the issues involved in writing unexploitable code. Two common attacks are buffer overflows and the double free attack. Since I'm not out to write a how-to on cracking security, I won't cover the details of exploiting these attacks any more than you need to know to avoid them. Instead, I'll talk about practices you can use to prevent them.

A buffer overflow occurs when you allow the user to enter more data than your program was expecting, thereby allowing arbitrary modifications to memory. Due to the way the stack is set up, an attacker can write arbitrary code into memory. This is how the Morris Worm worked, and it's how thousands of exploits since have worked. When functions are called, both the memory to store the variables declared in the function and the memory to store the arguments to the function are pushed onto the stack as part of a "stack frame". Here is a rough picture of what the stack frame would look like for a function call:

[memory for variables in the function][FP][ret][function arguments]

First, memory is allocated for variables declared in the function. Then the frame pointer, FP, which is used to address variables within the stack frame, then the address to return to after the function call, ret, followed by the arguments to the function. The gist of this attack is that on the stack, for every function call, the ret pointer indicates where in memory to jump once the function has finished executing. In normal execution, this should be back to the calling function. However, in some cases, if the program allows overflowing the buffer stored in the memory allocated for the function, it's possible for an attacker to set this value to point to an arbitrary region of memory into which the attacker has written executable code. (Often, this will actually be the buffer itself.)

How can a BUFFER OVERFLOW attack happen?

When you declare arrays in C or C++, you get a specified amount of memory to work with. This memory, on many systems, is placed before the pointer to the return site (where the function should return to after executing).

For instance, you might declare an array of 100 bytes:

```
char memory[100];
```

This is all well and good. But what if you really wanted to use 200 bytes? C will let you:

```
memory[150] = 'a';
```

There aren't bounds checks on the array, and the code might even work. (At least in some cases, you'll get a segmentation fault, but this will depend on whether or not the memory you're accessing belongs to your program or not. You might just overwrite other data in your portion of the stack.)

But you know not to just play with memory you haven't asked for, so you probably won't do things quite like that. Generally, what happens is that a function you call will overwrite the memory instead.

You might use a function such as gets() or strcpy that isn't aware of how much memory you asked for, and consequently, how large an array you have space for. This is particularly a problem for standard library functions that work on NULL-terminated strings such as strcpy, strcat, etc. Since they rely on finding a terminating character, if the string being worked with is too long, they'll happily overwrite the end of the buffer, and if you had declared the memory as an array, you might end up overwriting data on the stack. This is what is referred to as "smashing the stack".

Other dangerous functions include scanf and sprintf for similar reasons as gets.

What should you use instead? In place of gets() or using scanf to read in a string, use fgets()

char *fgets(char *buffer, int size, FILE *stream);

fgets takes a size -- make this the size of your buffer and it will read in up to size bytes into buffer from the file pointed to by stream. So, if you want to read from standard input (stdin) in order to replace gets:

char buffer[10];

fgets(buffer, sizeof(buffer), stdin);

This will allow the user to input up to 10 bytes for use in buffer. It will, however, add a '\0' terminator to buffer. Doing so would, of course, write off the end of the array. This means that you have to add the NULL terminator manually if you want to use functions such as printf that rely on it.

Note that sizeof(buffer) works because buffer is an array; you cannot do the same thing using a char* that you dynamically allocate using malloc or new.

In place of strcpy or strcat, use the corresponding strncpy or strncat functions that take the length of data to be copied.

char *strncpy(char *destination, const char *source, size_t n);

Using strncpy will result in at most n bytes being copied from source to destination, and strncpy will return a pointer to destination. For instance, if you know that destination can hold only 20 characters:

char destination[20];

char *source = "This is a particularly long string";

strncpy(destination, source, sizeof(destination));

This will copy only as much of source as can fit in destination and return a pointer to destination. You should be aware that strncpy will not automatically append a null terminator, which means that you can go from a regular, null-terminated string to a non-null-terminated string if you try to copy a string that won't fit into the destination.

char *strncat(char *destination, const char *source, size_t n);

The strncat function copies up to n characters from source onto the end of destination, starting from the null-terminator character of destination. Again, if you run out of space in destination before reaching the end of source, you won't get a null-terminated string back. And, to replace sprintf, you can use snprintf

int snprintf(char *destination, size_t n, const char *format, ...);

which will only write a string of n bytes into the memory pointed to by destination, protecting you from an attacker's writing arbitrary data into your string when you do something like

sprintf(dest, "The user entered: %s", user_input_string);

which allows the result stored in dest to be as long as is necessary to store the user_input_string.

Although we've only looked at examples where the size of memory was allocated at compile time, and consequently ended up on the stack, similar problems apply to memory allocated during program execution. Although I don't know of a way an attacker would be able to change the flow of control by modifying memory allocated on the heap, simply by changing variables an attacker could cause problems (imagine having a username field that your program uses for access control, and that an attacker can find a way to change that memory).

Double Free Attack

Another, more sophisticated attack is the double free attack that affects some implementations of malloc. The attack can happen when you call free on a pointer that has already been freed before you have re-initialized the pointer with a new memory address:

free(x);

/* code */

free(x);

This shouldn't ever need to happen in your code. The easiest way to avoid it is simply to set your pointer to point to NULL once you've freed it:

free(x); x = NULL;

```
/* code */
free(x);
```

Since free(NULL) is a valid operation that doesn't do anything, you're safe. Of course, this doesn't protect you from code that frees memory without telling you (or without making it obvious that memory is getting freed). One way of detecting these types of problems is to use a memory error detector such Valgrind to ensure that you only execute valid operations. (Valgrind will also detect use of initialized memory, use of invalid memory such as is the case with buffer overflows, and memory leaks.) Since the double free problem is one that should show up more or less regardless of user input -- you don't need a malicious attempt to overflow the buffer to test for double frees -- Valgrind is a good tool for finding these bugs. Of course, this isn't always the case. For instance, a double free vulnerability in the zlib library (CERT Advisory) required a certain type of user input to even cause free to be called twice on the same memory.

This bug is harder to exploit than potential buffer overflows, and it also relies on a particular implementation of the memory allocation system. Nevertheless, it's important to ensure that your code correctly frees only valid blocks of memory.

Brain teasers

1. Tom's mom had three children. The first was named May, the second was June. What was the third childs name?
2. The manufacturer doesn't want to use it, the buyer doesn't need to use it and the user doesn't know he's using it. What is it?
3. The word CANDY can be spelled using just 2 letters. Can you figure out how?
4. Bill bets Craig $100 that he can predict the score of the hockey game before it starts. Craig agrees, but loses the bet. Why did Craig lose the bet?
5. What is the next 3 letters in this sequence?
   o t t f f s s _ _ _
6. FOUR is HALF of FIVE.
   Is this statement True or False?
7. A woman shoots her husband.
   Then she holds him under water for over 5 minutes.
   Finally, she hangs him.
   But 5 minutes later they both go out together and enjoy a wonderful dinner together.

   How can this be?
8. What do these 3 have in common?
   Superman
   Moses
   The Cabbage Patch Kids
9. What is black when you buy it, red when you use it, and gray when you throw it away?
10. Can you name three consecutive days without using the words Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, or Sunday?

Answers

1. Tom.....Tom's mom had three children, June, May, and Tom.
2. A Coffin
3. The answer: C and Y
4. Bill said the score would be 0-0 and he was right. Before any hockey game starts, the score is always 0-0.
5. "e n t " They represent the first letter when writing the numbers one thru ten.
6. It's True. The Roman Numeral FOUR (IV) is in the middle of the word Five: F(IV)E
7. The woman was a photographer. She shot a picture of her husband, developed it, and hung it up to dry.
8. They were all adopted!
9. The answer is Charcoal. In Homer Simpson's words: Hmmmm... Barbecue.
10. Sure you can: Yesterday, Today, and Tomorrow!

Q) Imagine that you are in a boat, in the middle of the sea. Suddenly you are surrounded by hungry sharks, just waiting to feed on you. How can you put an end to this?

A) Stop Imagining!

"The staff are healthy enough, it's the computers that keep getting a virus!"

WE LOOK FORWARD FOR YOUR FEEDBACK