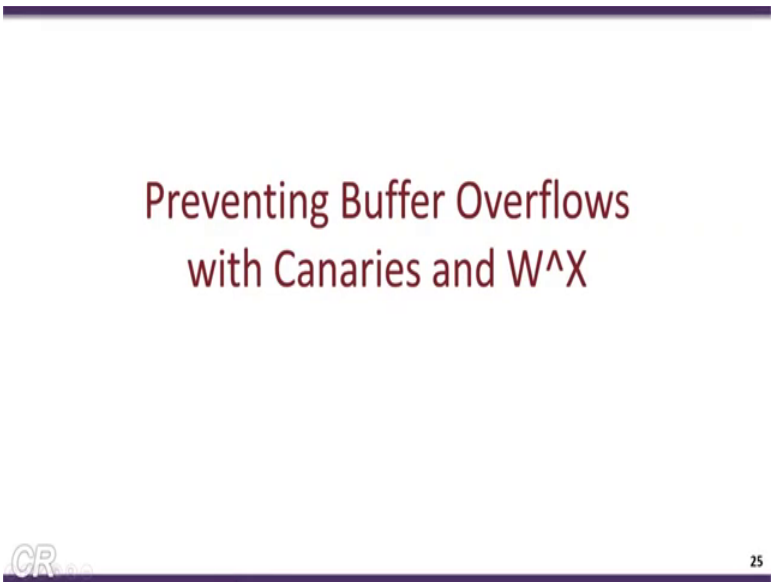
Information Security-5-Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
W2_1
Preventing Buffer Overflows with Canaries and W^X

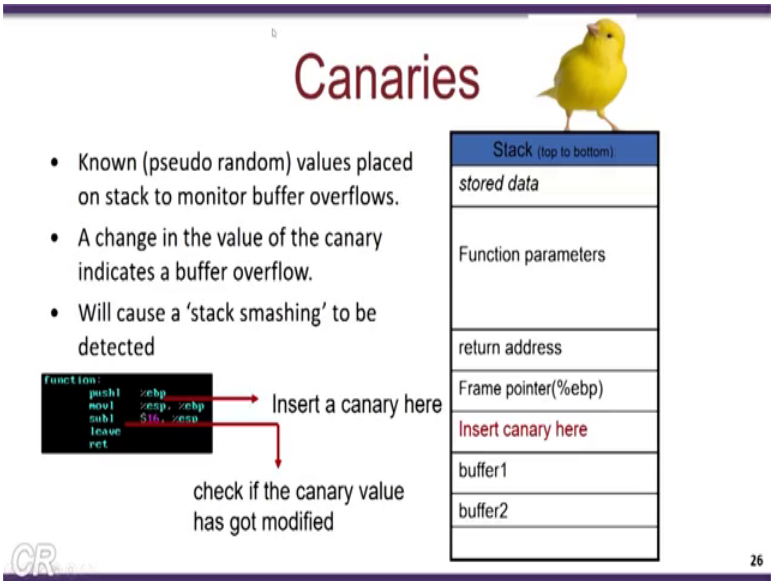
(Refer Slide Time: 00:12)



Hello and welcome to this lecture in the course for Secure System Engineering, in the previous lecture would actually look at a buffer overflow in the stack and we seen how that vulnerability or rather that buffer overflow can be exploited it can be used to support execution and an attacker could write a payload and force that payload to execute. So essentially what the attacker would do was overflow the buffer so that the return address which is present on the stack is over written.

Now the return address is over written with another location on the stack and in that location the attacker has written a payload and this payload would then execute. In this lecture we will look at two prevention techniques, one is called canaries while the other one is called the NX bit or the non-executable stack both this different techniques are very popular and have been incorporated in all compilers and systems that are in use today. So let us start this lecture.

(Refer Slide Time: 01:31)



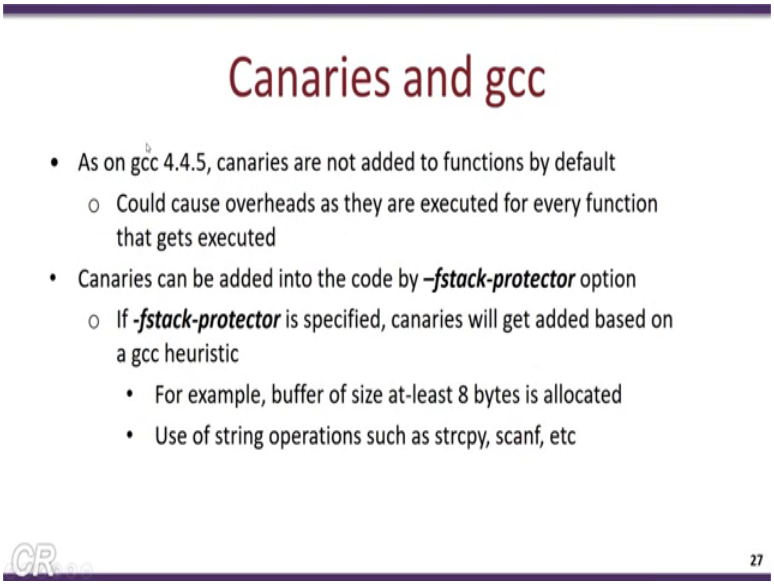
So lets talk about canaries, so canary is a compiler defense mechanisms essentially what the compiler does is that whenever a stack frame gets created the compiler could insert a canary in the stack. So note that in this particular figure we show the stack of the program as we have seen before in the previous lecture the stack would contain the function parameters it would contain the written address then it would have a frame pointer and then the local variables for that invoked function. So essentially what the compiler would do is insert a canary over here.

Now what is a canary? Now a canary is essentially a random number which the compiler fix from a particular random store and inserts it over here. Now whenever if any of this two buffers overflow that is buffer 1 or buffer 2 overflows and it crosses the canary then the canary value will be modified. Now at the end of the function the compiler would detect that there is a change in the canary value that is it would throw an error that and that it will throw an error stating that a stack has been modified.

So if you look at this particular small function over here which essentially creates the stack frame it pushes the stack frame pointer EVP on the stack that corresponds to this and then there is a changing of stack frame that is the current stack pointer is moved to ace pointer and then the stack pointer is decremented by 16 bytes to give space for the local variables. Now this is a function without canaries. Now in compliers which supports canaries additional instructions are present at this particular point where instructions are present to add, insert a canary over here and

just before the return statement more instructions are present so that so as to ensure that the canary has not ben modified.

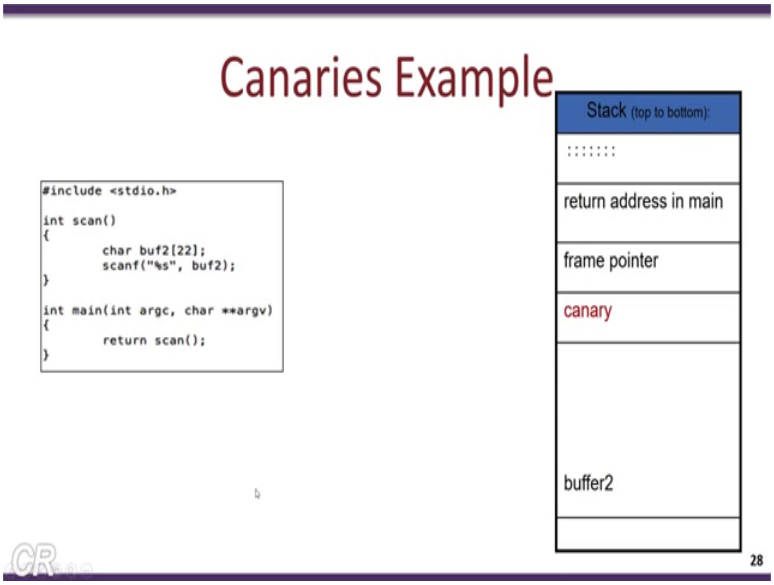
(Refer Slide Time: 04:01)



Most of the compilers support canaries, the compilers that we are using in this course that is GCC also has support for canaries. So in some compilers the canaries are enable by default while in other compiler other versions of the compiler you would have to give an explicit compilation flag that would enable canaries to be inserted within functions. Some compilers also use heuristics to insert canaries for instance if a compiler finds that in a function there is a potential for a buffer overflow only then the canaries are inserted.

In order to enable a canaries in versions of GCC which do not support canaries by default you could add these compilation option minus f –stack protector. So adding this minus f-stack is an indication to the compiler to add canaries to the functions.

(Refer Slide Time: 05:02)

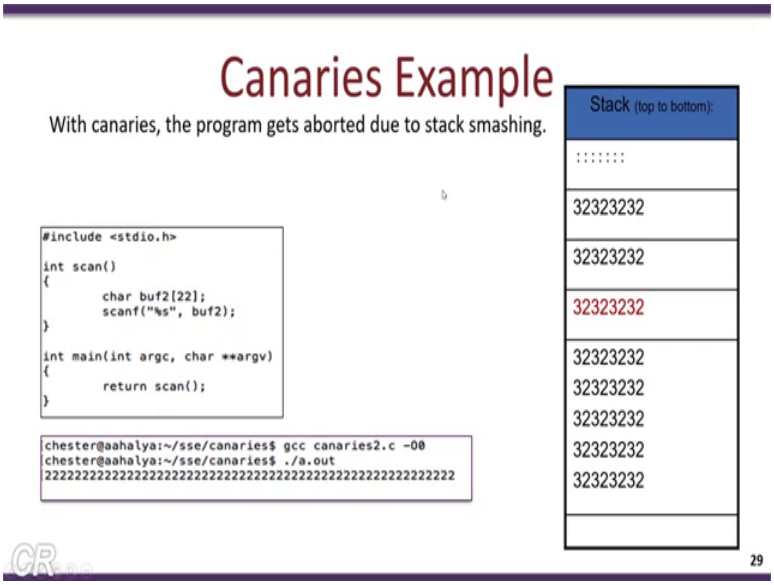


So let us see how the canaries actually work internally with a small example. So we will take this particular example where there are just two functions one the main function and then another function called scan and the main function is just invoking scan and then returning.

Now in the scan function we declare a local buffer of 22 bytes and as we know since it is a local variable this array that is allocated in the stack. So then now we invoke this scan f function which reads a string from the user. So note that this particular scan f function is a potential for a buffer overflow the reason is that the user could enter a large string which is much larger than 22 bytes and therefore buffer 2 can overflow. Now let us look at the stack, so when the main function invokes the scan function this is how the stack is going to look like.

So there is the written address pushed into the stack this corresponds to the address soon after the scan instruction in the main. Then there is the previous frame pointer pushed onto the stack which will enable the frame to change when the scan function gets returned then the compiler would insert a canary and there would be 22 bytes of space for the buffer 2 to be present.

(Refer Slide Time: 06:36)

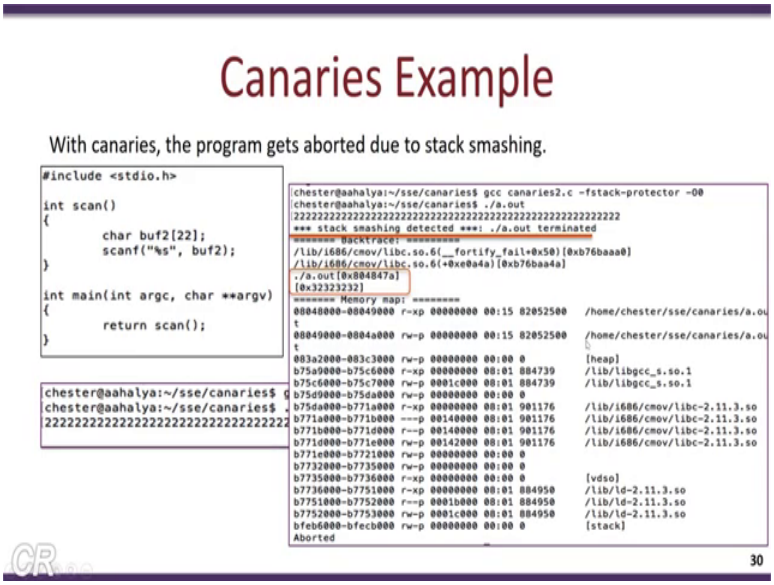


Now let us see what would happen when we run this program and give a large input which is much larger than 22 bytes for example we compile this particular program like GCC canaries 2 dot C minus o 0 now in this particular version of GCC that I have used for compiling canaries are inserted by default.

However this may not be the case for all GCC versions in which case you have to put minus f-stack projector as an option during compilation. Now when you run this particular program main called scan, scan call scan f and it prompts the user to enter a string so what we have done here is we have entered a very large string much larger than 22 bytes as you know what is scan f does is that it would fill the buffer 2 with this particular string. Now the ASCI value for 2 in hexadecimal notation is 32. Now what happens here is that this value of 32 that's filled in that stack is starts off with filling buffer 2 and since we are exceeding the 22 byte limit of buffer 2 there is a buffer overflow and we note that the canary value gets changed.

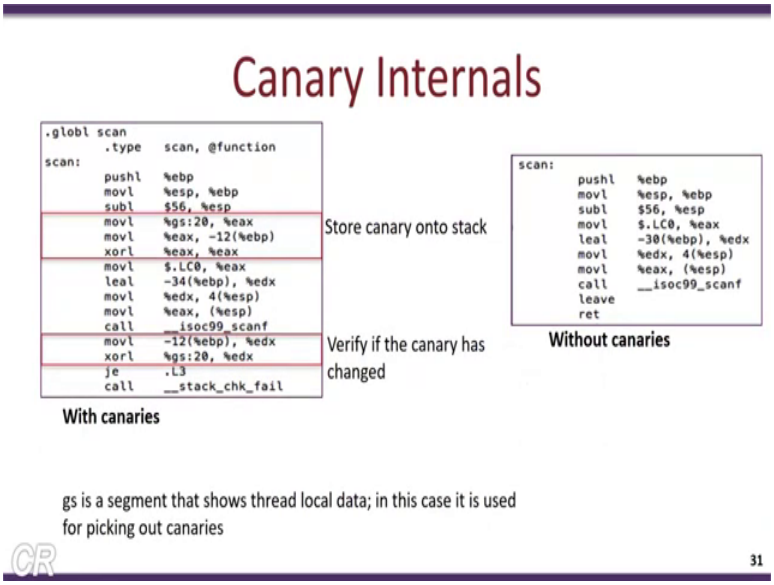
So whatever canary was there present is now overwritten with 32, 32, 32 and 32 similarly the other data on the stack including the written address and the frame pointer gets overwritten with 32's. Now when the scan function returns what happens is that the compiler has added code that detects whether the canary value has changed.

(Refer Slide Time: 08:28)



As a result we will get an output which looks like this, you see that there is a stack smashing detected and the program is downloaded. So note that there is one thing over here it shows that the program has terminated at this particular location 0 x 804847A and the value of the canary which has been modified was 32, 32, 32, 32 this is indeed what was present in the stack due to the buffer overflow. So the result of the stack smashing detection would also provide the memory map of that particular program at the point of termination.

(Refer Slide Time: 09:07)



So now let us dwell a little more deeper into what actually happens with canaries. So over here on this part of the slide shows the assembly code for scan without canaries enabled while on the left side shows the corresponding assembly code that gets complied with canaries enabled. So you notice over here that without canaries the number of instructions are very less, note that as usual there is stack frame that is created and then (there are) some parameters which have been to be set for scan f and then there is an invocation to the scan f, this particular instruction called ISO C99 scan f would invoke the scan f function from the library and then there is a return from the function.

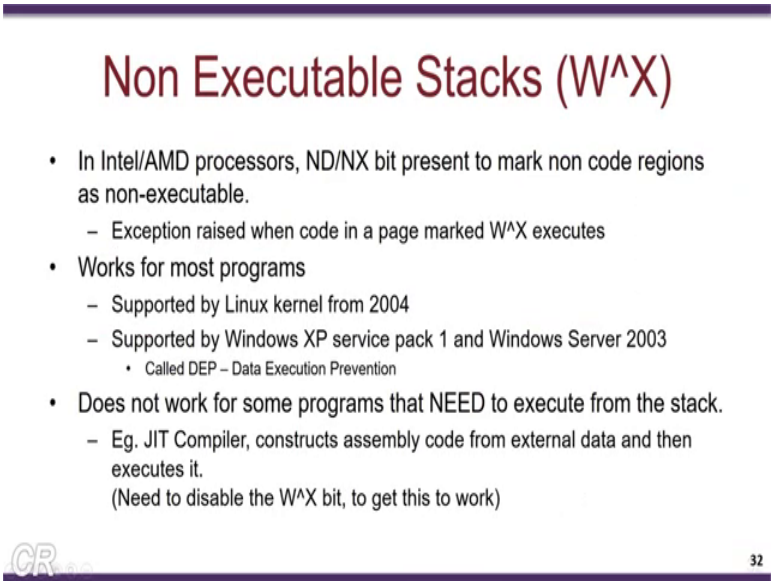
Now when canaries are enabled there are a few extra instructions that gets added, now precisely we have three instructions at the start of the function and two instructions that gets inserted at the end of the function. Now this set of three instructions essentially loads at random canary into the stack so the random value which is going to be present in the canary is obtained from this particular location GS colon 20 that is in the segment pointed to by the GS and at an offset 20 the random data which is present over here which is essentially that segment has is a random state so we are taking some random data from there moving it to the EAX register and then storing the contents of the EAX register into the stack.

Now at the location EVP that is a frame pointer minus 12 bytes is where the canary location is present. Now at this location we are storing the contents of EAX which is the random value for

the canary then we are having an EX-OR instruction which essentially makes the EOX register zero. Now we have the rest of three instructions as was here before and at the end just before the return from the function there is a check which is done to detect whether the canary has changed or not. Now the check essentially would load the canary value from the stack that is at location EBP minus 12 into the EDX register and then it would check whether the contents of the canary has changed.

So this is done by checking whether the EX-OR of the original data present in GS colon 20 is the same as that in the EDX register, it would that the now if it is the same it would mean that there is no change in the canary and the function return successfully. On the other hand, if at this point we detect that the canary has indeed changed it would mean that the EX-OR value would return something which is non-zero and then there would be a call to this particular function called stack check fail. Now stack check fail is a built-in function which essentially would print out this particular output which tells you the exact information about where the stack smashing was detected and so on.

(Refer Slide Time: 12:50)



And other defense mechanism is implemented in the hardware by the processor manufactures so this is called the non-executable stack or the W ^ X bit. So what this means is that for a particular segment for example the stack segment one can either write to that particular segment or execute from that segment. So for example if you can write to a particular segment in would mean that

you cannot execute from that segment. On the other hand, if you can execute a segment we cannot write to that particular segment. So let us see why this thing would work.

So recollect that in the previous lecture when we looked at the buffer overflow and how the attacker wrote his payload in the stack and then executed that particular payload is that the attacker was able to inject code by writing to the stack and then execute in the stack. Now with this non-executable stack one of these two is not possible so if you write to the stack it would imply that you cannot execute from that stack. So both Intel and AMD processors support these non-executable stacks.

On Intel processors this is called as the NX bit while in the AMD processors this is known as the ND bit. So the NX bit or the ND bit is present to mark the non-code regions as non-executable thus the stack of the particular program would be having its NX bits set which would mean that you could write to the stack but you cannot execute from that stack.

Now by chance if let us say the payload is trying to execute from that particular stack then an exception get raised and the code will terminate. So the NX bit support from the OS perspective has been supported from the Linux kernels since 2004 and also Windows XP service pack 1 and Windows Server 2003 so this is called the data execution prevention. Now we would note that this is a very efficient way to prevent the attack, the reason is that all that is required is just 1 bit for a page and this bit incorporated in the page directory and page table for that particular process and it is just that single bit which is sufficient to actually prevent the attack.

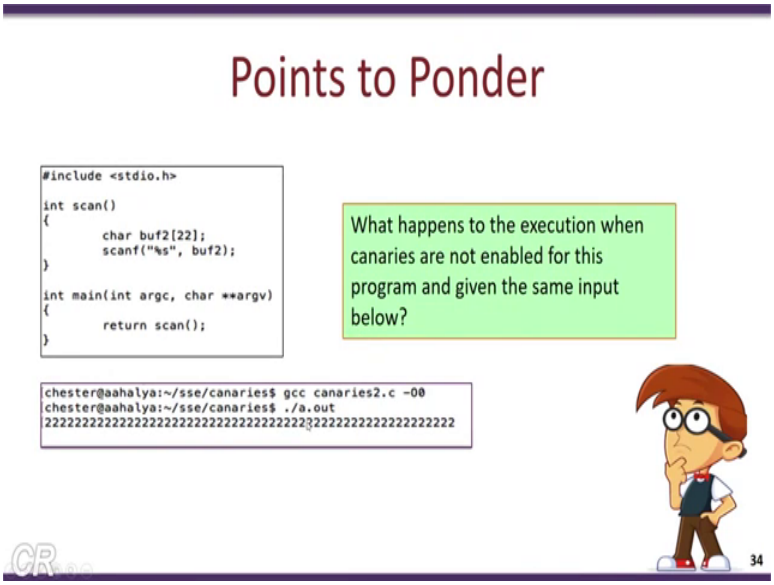
However there are several non-malicious programs which actually would need to execute from the stack for example the JAVA just in time compiler would construct assembly code based on some external data and then execute it. Now for such applications the code gets constructed on the stack before it gets executed. So this application would not complete if the NX bit gets set. So in order to actually run this particular applications on a processor with NX bit enabled it would require you to disable this NX bit before it this particular application can work. Therefore, why this is a very efficient way to prevent this buffer overflow attacks, there is also a downside present the certain types of applications do not complete.

(Refer Slide Time: 16:32)

So both this defense mechanisms the canaries as well as the NX bit are incorporated in most modern systems. So quite likely if you try to run this particular code on your latest for example Ubuntu systems you would get stack smashing getting detected and this particular code will not run. However, in order to make it run we would have to disable this particular defenses and the way to do that is by compiling this particular program with certain option so as to disable this stack protection and the NX bit.

The way to do it is by specifying minus f no stack protector which would disable canaries and minus z execute stack which would permit execution from that stack. When this is done we would get an executable a dot out and this a dot out should successfully run on your machine. Now you can refer to this code in this particular directory and compile it with these options and then see this particular program executing and have the payload running and the shell getting created successfully.

(Refer Slide Time: 17:52)



So before we complete this lecture there is some points that you can think about. Now when we consider this particular example of the main and scan and we consider that when use scan f we have given a very large input of all 2's. Now what would happen to the execution when the canaries are not enabled for this particular program? That is in this very specific program suppose you have compiled it without canaries being enabled that is by example using the minus f no canaries option during compilation, what would be the result of this particular program? So this is something to think about until the next lecture, thank you.