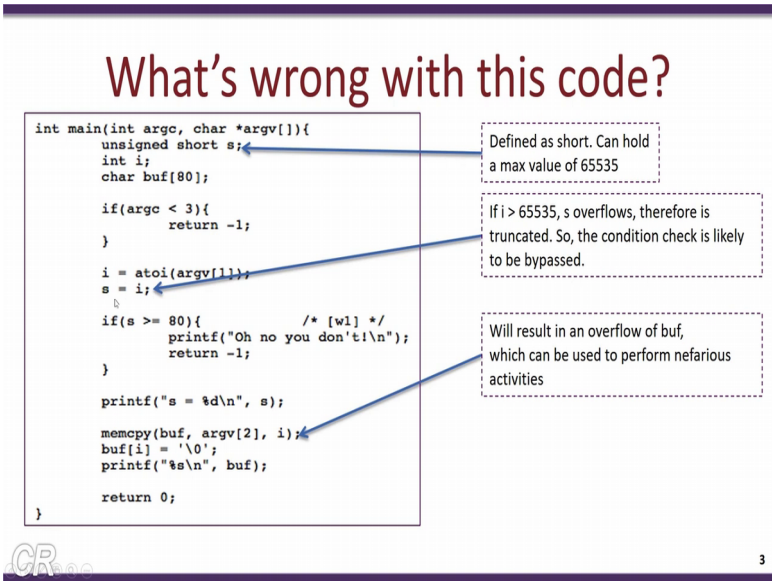
Information Security 5 Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology Madras
Integer Overflow Vulnerability

Hello and welcome to this lecture in the course for secure systems engineering, in the previous lectures we had looked at buffer over read vulnerabilities and then we also looked at format string vulnerabilities. In this lecture we will look at another form of vulnerability known as integer over flow vulnerabilities,

(Refer Slide Time: 0:39)



So let us start with a small example, so let us consider this particular program over here and what we do in this program is take 2 arguments argv1 and argv2. Now argv1 is essentially an integer number and therefore we convert argv1 to this value of i then we copy this value of i to s. We check whether s is greater than equal to 80, if so then we printf 1 statement 'Oh no you do not' and then return minus 1. If s less than 80 then we copy the contents of argv2 into this particular buffer we add slash 0 to that particular buffer at the end of this buffer and then print the particular buffer okay so the expected behaviour would look something like this.

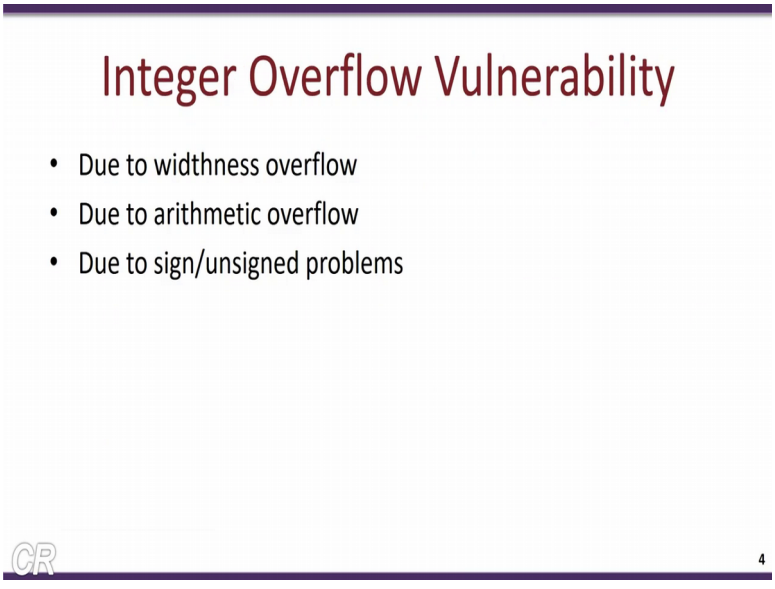
When we run this program which is known as width1, we specify 5 and we specify argv2 as hello then what we did is that hello gets printed on the screen. On the other hand if you specify argv1 as 80 then it is caught by this if statement over here we get 'Oh no you do not' gets printed on the screen and there is a return minus 5. The reason we create this check is that we have this local buffer buf which is of 80 bytes. Now this C program seems pretty

straightforward it seems to be working fine but there is a problem, the problem is that we have defined s to be unsigned short and i to be of integer.

As you know short values can hold the maximum value up to 2 power 16 minus 1 which is a 65535. On the other hand integer which is typically 4 bytes could store up to a 32-bit value thus if i is greater than 65535 there is an overflow that occurs and therefore s gets truncated for example if i is equal to 65536 right then s would become 0 because s holds only 16 bit value and s is of short and therefore s would become 0. So what happens in such cases is that we can trick this program to print more than what is expected, so for example suppose we specify that argv1 in other words i is say 65537 this will cause s to get truncated and have the value of 1.

Now s has a value of 1 which is definitely less than 80, so this test will fail and we would enter this part and here you notice that we are copying argv2 into buf with respect to i and not s thus we would copy 65537 bytes from argv2 into buf so this may create a fault and the program may crash but it serves the purpose to actually understand how difference between a shot and an int and seemingly denying statements such as this could actually create havoc in your program.

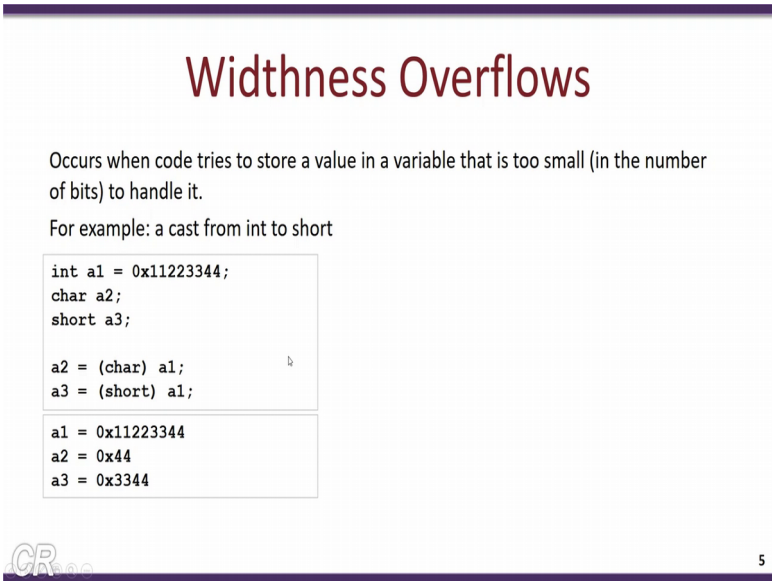
(Refer Slide Time: 4:27)



So there are 3 different types of integer overflow vulnerabilities the first is due to widthness overflow second is due to arithmetic overflow while the third is due to sign and unsigned problems, so we will look at each of these cases with some examples and see what could go

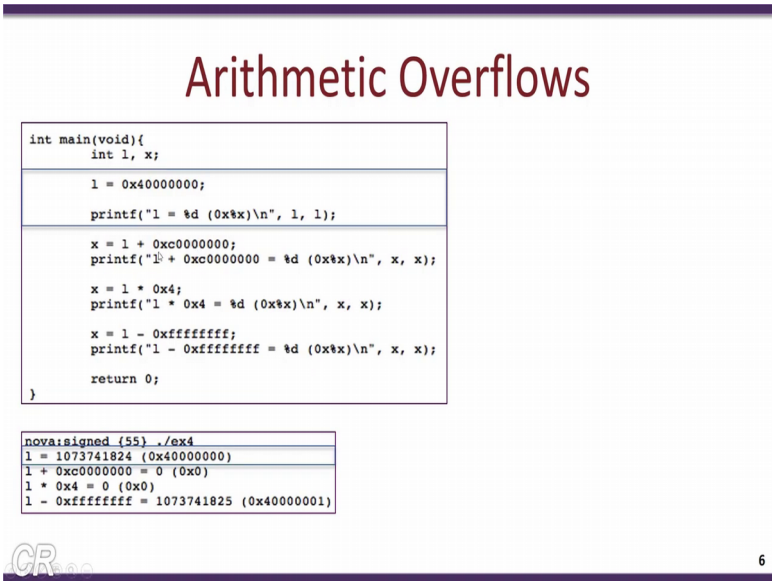
wrong in the program and how an attacker could be able to do something which is not expected of the program.

(Refer Slide Time: 4:57)



So let us start with widthness overflow, so this is very close to the example that we have seen and it is mostly related to when we try to do typecasting, so for example if we have defined an integer over here and initialised it to 11223344 what we do know is that this integer will take will occupy 32 bits on the other hand this character as well as the short would occupy 8 bits and 16 bits respectively, so therefore when we actually have statements such as this where we try to typecast a1 to this character a2 or typecast a1 to the short int a3 it would result in truncation, so if we now print the values of a1, a2 and a3 we will get this values. So not that a2 and a3 get truncated to 8 bit and 16 bit respectively.

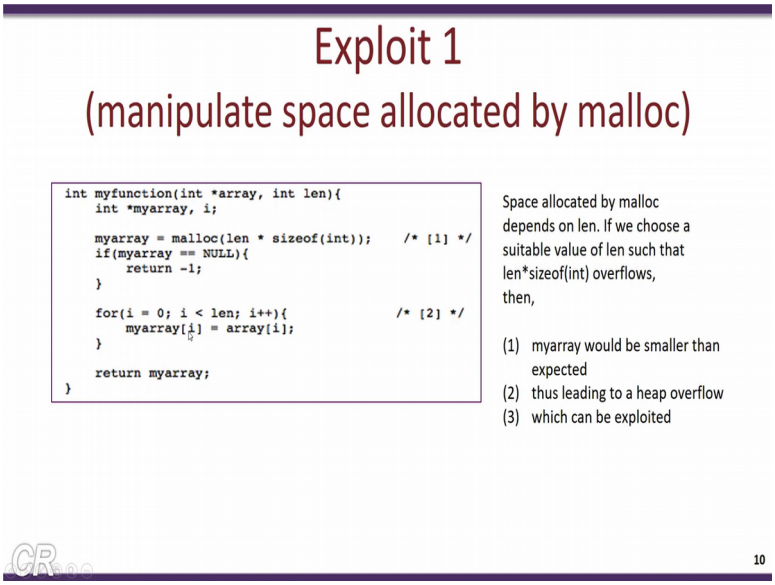
(Refer Slide Time: 6:00)



Next people look at arithmetic overflows and to understand arithmetic overflows we look at this program, we define an integer l and initialise l to 0x4 followed by 7 0s and when we do print this value of l in decimal and hexa decimal notation we get what is expected. Now let us see when we do some operations on this l, so let us say we take l and add 0 at c followed by 7 0s what we obtain when we print the value of x is 0 and this happens because 0x4 followed by 7 0s plus 0x7 followed by 7 0s would be a value of 1 followed by 32 0s and since x is also defined as an integer it can only hold 32-bit value and therefore the results truncated the 33rd bit and about would be ignored and x would get a value of 0.

So similar kind of arithmetic overflows can be also seen when we do multiplication, so for example you take l multiply it by 4 and when we do print it you will see that the result is 0. Another example is when we take L and subtract 0xf 7 fs what we get is something which is not expected for example when we do subtract we get 0x4 followed by 6 0s and then a 1, the reason for this is that in sign notation this value of 0x all fs is taken as minus 1 in two's complement notation therefore it is l minus of minus 1 which is plus 1 and therefore your result would be l plus 1 which is this value.

(Refer Slide Time: 8:12)

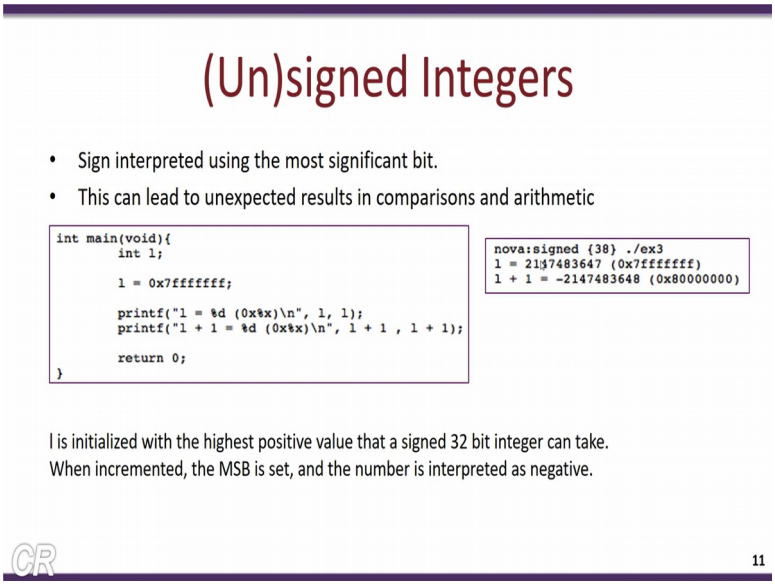


Now let us look at some simple exploits which make use of the arithmetic overflow vulnerabilities, so will start with this particular example where we will show how we could manipulate the space allocated by malloc. What we do is simple the function takes 2 parameters and integer array followed by the length and then we malloc length into the size of integer, so this as you would know would be typically how you would create a malloc. Since we want an integer array and each integer is of 4 bytes therefore we typically do len into the size of the integer and we check whether malloc was done successfully and then we copy array into myarray.

The problem with this function is that the input length is set by the user or rather by the user who is actually invoking this function, so therefore the user could actually give a very large number for len such that when you multiply len by size of int which is typically 4 bytes then there could be an arithmetic overflow and the value returned the value passed and then the value passed to malloc could be a very small number.

Thus my array gets allocated with a very small memory size. Now what would happen over here in this for loop as we see that the for loop runs from I equals to 0 to i less then len. Now len we have specified as a very large number and therefore we would obtain a buffer over low for my array of i, so what we see over here is that we have been able to invoke malloc and being able to overflow the number of bytes requested to malloc thus the array that we obtain is very small and we could cause an buffer overflow to my array, so as we have seen before when we create such buffer overflows it can thus be exploited.

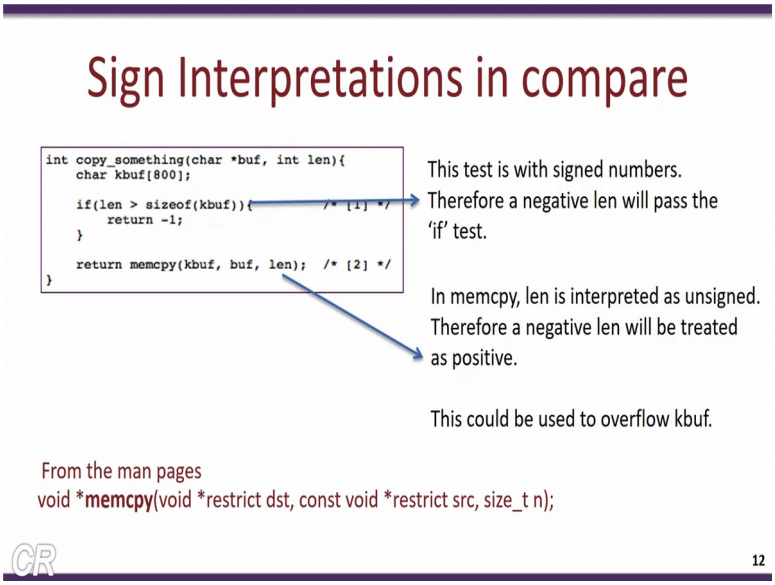
(Refer Slide Time: 10:26)



The next thing we look at is unsigned integers and the vulnerabilities that can be caused by due to sign and unsigned integers, so as we know signed integers are interpreted using the most significant bit. If the most significant is set to 1 then the number is set to be a negative number. If the most significant bit is set is to 0 then the number is interpreted as a positive number.

This can lead to unexpected results especially when you are doing comparisons and arithmetic, so let us take this very small example so what we have done is that we have defined the integer l to be 0 x 7 followed by 7 fs, so this is the largest positive sign integer that can be represented with a 4 byte integer value, so what we do is we add 1 to this value of l and see what would happen, so when we do had 1 what we see is that the number surprisingly changes from a very large positive number to the smallest negative number, so this result is not always expected and therefore is a vulnerability which could lead to creation of exploits.

(Refer Slide Time: 11:52)

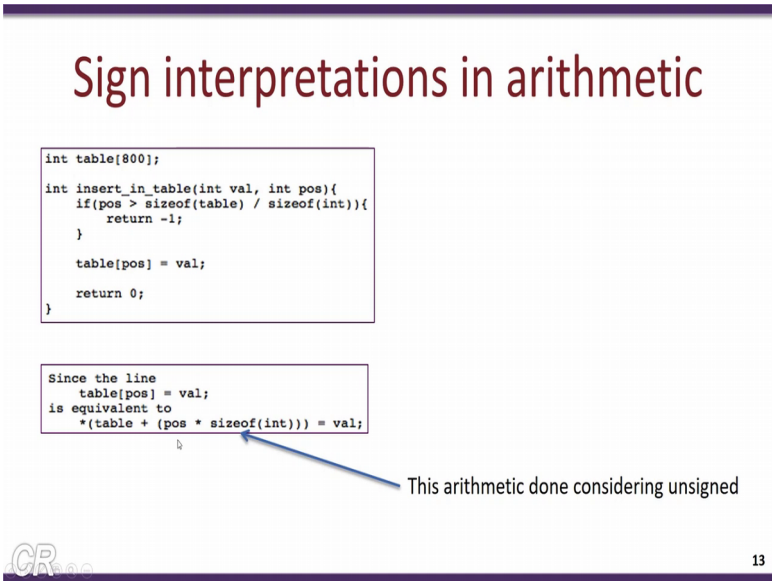


So let us look at a small vulnerability with sign interpretation that involve comparisons and then copying, so let us take a look at this function where what we do is we take the parameter length which is passed as input, check whether len is greater than size of kbuf. Now kbuf is a local over here and comprises of 800 bytes of data and if it is len is less than or equal to this 800 bytes then we do not go into the if statement but rather do a memcpy where we copy buffer to kbuf, so what is the vulnerability in this particular function?

The 1st thing you would not is that len is defined as int right it is a signed integer on the other hand if you look at the man pages of memcpy what we see is that the 3rd parameter that this over here size underscore t n which corresponds to the len over here is actually defined as unsigned, so we have internally a (())(13:10) or size underscore t to be an unsigned integer, so what we see over here is that that an implicit type casting from a signed integer to an unsigned integer.

So for example if we give len to be minus 1, minus 1 is definitely less than 800 and therefore this if returns falls and this return statement is not executed but rather we would have a memcpy getting executed. Now when memcpy executes since it expects the 3rd parameter to be an unsigned integer therefore there is an implicit type casting of len from signed to unsigned. Now internally the signed integer for minus 1 is this very large value of 2 power 31 minus 1 and therefore what we are getting is a buffer overflow. What could happen is that we would have a large buf which could be a much larger than 800 bytes getting copied into kbuf which is strictly restricted to 800 bytes thus we could get a buffer overflow which could be then used to create exploits.

(Refer Slide Time: 14:27)



So let us look at another example of this particular function over here where essentially we want to fill one particular entry in this particular table defined as global table of 800 entries with some value, so we pass the value as the first parameter and the position in the table as the 2nd parameter, so what we want to do eventually is to say that table of pos equal to val, so before doing this we do the customary checks, we check that if pos is greater than size of table divided by size of int.

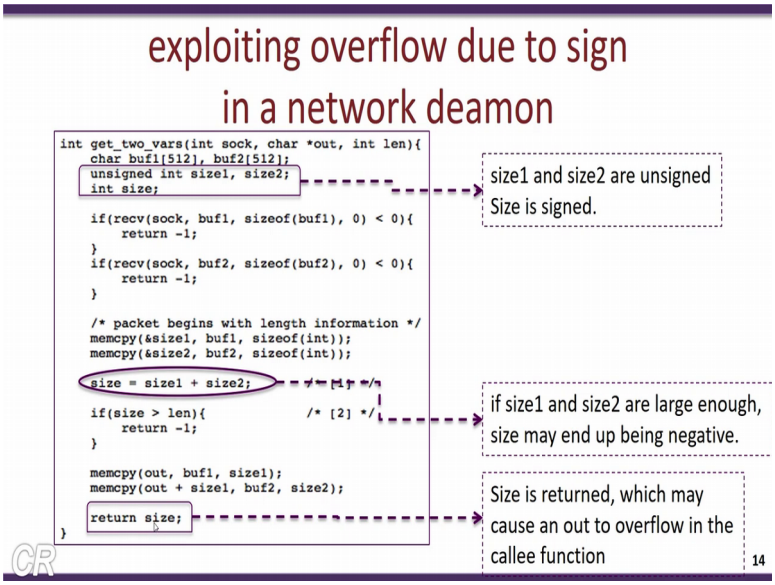
So size of table would be 800 times 4 divided by size of int so these 2 together would be 800 we to this check and we checked whether pos is a valid index for this table, if so only then we copy val into pos, so one thing to note is that this statement table of pos equal to val is actually executed as follows, so essentially this statement table of pos equal to val is interpreted as star table plus pos in to size of int equal to val.

Now what we do is we take pos multiply it by size of integers 4 add that to the starting address of table and at that particular location we fill in the value of val. Now the vulnerability is at this point, problem with this code is that pos is defined as an integer and therefore can take both positive as well as negative values. If we give a negative value for pos for example let us say pos is equal to minus 1 then this if statement would have minus 1 greater than 800 and definitely this particular if statement would be false.

On the other hand when we come to this statement over here a table of pos equal to val over here pos is taken as an unsigned value so the minus 1 which is pos is interpreted as a positive value and therefore we would have a table added to a very large positive value which is a

causing a buffer overflow, so the val could be stored in a particular location which is much further away than the actual size of the table.

(Refer Slide Time: 17:13)



So let us look at another example of the integer overflow vulnerability due to sign in a network process or daemon, so an example of this is given in this function over here so what this function does is that it accepts 2 packets 1 is buffer 1 and buffer 2 so these 2 packets are obtained through sockets and then what it does is that assume that first 4 bytes of each of these packets contains the size of the packet so for example buffer 1 would look something like this way the 1st 4 bytes would have the size and the remaining bytes would have the payload and similar structure is present for buffer 2 as well.

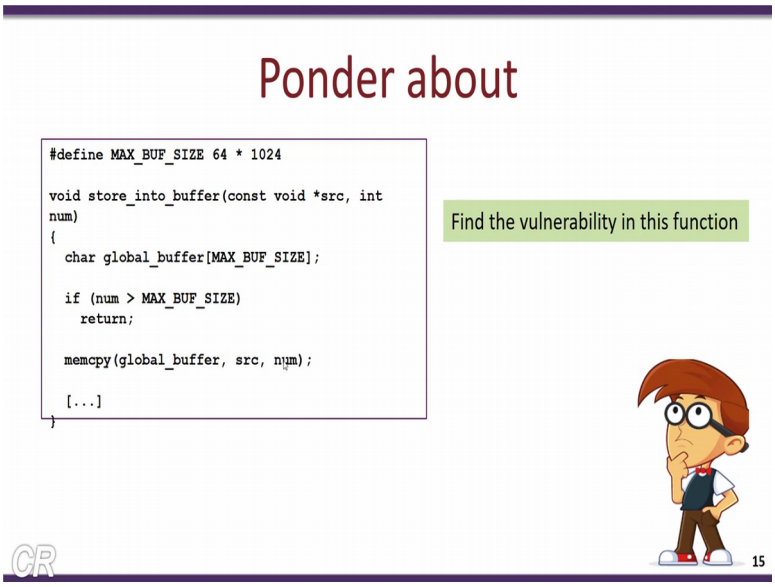
So therefore with these 2 memcpy instructions we obtain size 1 to be the size of the payload for buffer 1 and size 2 to be the size of the payload for buffer 2. Now we add these 2 things together to obtain size and then what we do is we would fill the buffer called out with buffer 1 followed by buffer 2, so essentially out is expected to be the concatenation of buffer 1 plus buffer 2. So next what we do is we store in the character pointer out we store both buffer 1 followed by buffer 2.

In other words out comprises of the concatenation of buffer 1 and buffer 2 and then we return the size. Let us see the vulnerability in this particular program 1st vulnerability you would notice is that size 1 and size 2 are set by the application which is sending these particular packets. The next thing you would notice is that size 1 and size 2 is defined as unsigned integers while on the other hand size is defined as a size integer. Now this operation over here

size 1 plus size 2 would give you size is a vulnerability, so it is on the right hand side we have unsigned integers while on the left-hand side we have a signed integer.

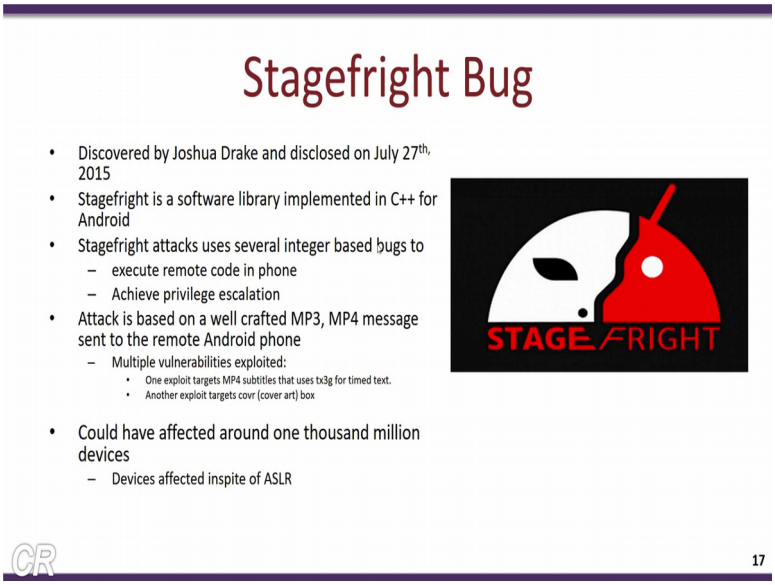
Therefore if size 1 and size 2 is large enough we may actually obtain size to be a negative value this size is what is returned by the function and since size can be a negative value it could result in an overflow in the callee function. What we would have is out to be a very large value comprising of buffer 1 and buffer 2, what could happen is that out could be a very large value because size 1 and size 2 are specified as a large unsigned integers on the other hand size is a the negative value as a result of this what would happen in this memcpy is that out could be a very large buffer because size 1 as well as size 2 is very large. On the other hand size could be a negative number as well, so this could result in a buffer overflow in the callee function.

(Refer Slide Time: 20:40)



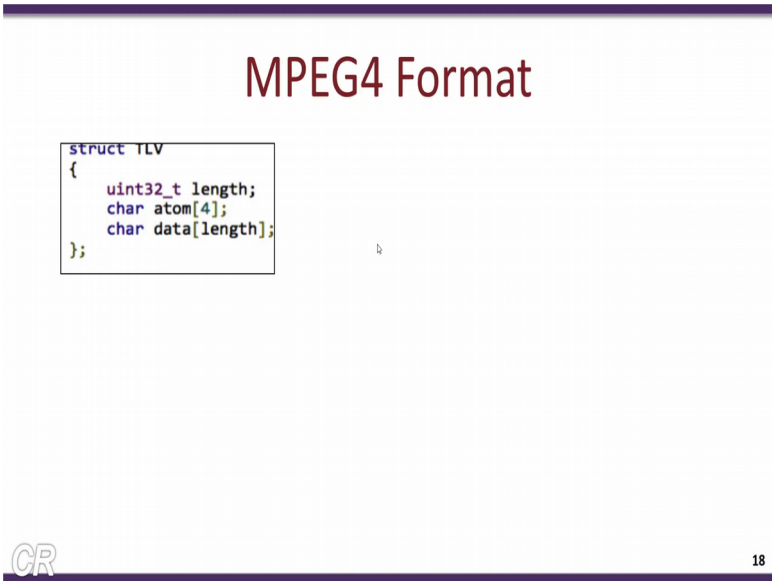
So this is a small exercise for you to do so in this function called store into buffer we have a local buffer of size max buf size. Max buf size is defined as 64 times 1024 if num which is passed as input is greater than max size then we return else we do a memcpy of source to the global buffer and the number of bytes we are actually coping is specified by num, so I would like you to find out what the vulnerability of this function is?

(Refer Slide Time: 21:18)



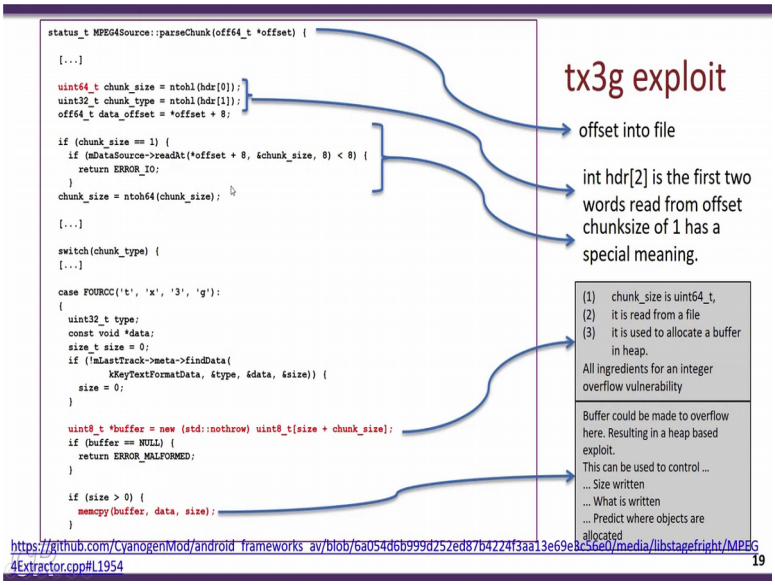
One quite famous malware which actually uses integer overflow vulnerabilities quite a bit was known as the stage fright bug, so this particular bug was discovered by Joshua Drake and was disclosed on July 27th 2015, so the stage fright software is essentially a software implemented in C plus plus for android applications and because of the bug could actually do several things such as it could cause like privilege escalation attacks on your mobile phone or it could also execute arbitrary code on the phone, so the essence of the bug is based on MP3 and MP4 files, so essentially what the attacker does is he creates well-crafted MP4 files which is then decoded by the stage fright library and then these MP4 files are designed so that it could trigger the vulnerability and then cause the payload to executed or it could cause escalation of privileges.

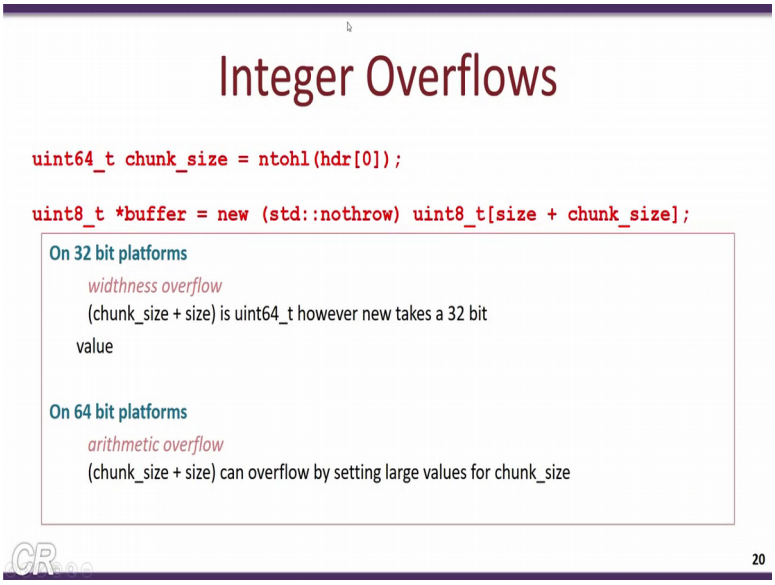
(Refer Slide Time: 22:36)



So the essential idea is based on this particular structure, so what we see over here is that this structure comprises of 2 arrays one is an array call atom and another array called data which is of size length. Now this particular structure is present in the MP4 file, so what you see is that an attacker could create an MP4 packet with a corrupted length value.

(Refer Slide Time: 23:07)





So this actually shows the broken code, so there are a lot of things which are involved so I will not go into the details of it. There were multiple overflow vulnerabilities that were exploited on 32-bit platforms essentially it was a widthness overflow that was exploited while on 64-bit android platforms it was an arithmetic overflow that was exploited. Now the integral statements in this function over here are these 2 here it is definition of chunk size which is taken from the header, so the header is present in the MP4 file and could be set by the attacker.

This chunk size is used to actually malloc and array of size plus chunk size, so this is array of size of unsigned int 8 which is essentially an unsigned character array, so the overflow that is occurring here is because of the fact that this header 0 comes from the MP4 file and could be manipulated by the attacker, so for example a large value of header could be set and it could cause widthness overflow or arithmetic overflows on various platforms. Thank you.