

Software Security Lesson Introduction

- **Software vulnerabilities** and how attackers **exploit them**.
- **Defenses against attacks** that try to exploit buffer overflows.
- **Secure programming**: Code “defensively”, expecting it to be exploited. Do not trust the “inputs” that come from users of the software system.

operating system can do for us. By the end of this lesson, hopefully, you're going to understand the importance of what we call secure programming, and will practice it when you write code.

Software Vulnerabilities & How They Get Exploited

- **Example: Buffer overflow** - a common and persistent vulnerability
- Stack buffer overflows
- **Stacks** are used...
 - in **function/procedure calls**
 - for **allocation of memory** for...
 - local variables
 - parameters
 - control information (return address)



Instructor Notes

Poor Software is the Biggest Cyber Threat

So, software is what controls our computer systems. We get things done by running software that we write, or somebody else has written for us. And, we're going to talk about exactly what are those bugs that actually turn into vulnerabilities. We're also going to talk about how exactly those bugs, or others get exploited.

So the vulnerabilities that we're going to talk about come because of memory overflow. Keep in mind memory overflow means that the amount of memory we have for a certain data type is not sufficient so the data type runs over the allocated space. And an attacker is actually able to exploit a program by inserting new code sometimes in certain part of memory. And also then directing or transferring control of that code to the instructions that the attacker has introduced in this memory that we're talking about.

Stack is the area where we allocate space for dynamically created data items or variables. So the most common example of when you use a stack is when you make a function call or a procedure call. When we do that, the variables that we need to execute the function code get allocated on the stack. So, as you make function calls, or procedure calls, for each call we create what we call a stack frame on the stack. This stack frame essentially you can think about, gives us the scratch pad or the memory that we going to need for the execution of this function. It's created when the function is called and it is discarded when the function finishes and returns.

So what exactly is stored in the stack frame?

- Part of this we're going to allocate space for local variables that are going to be used by the code defined by the function.

- Also parameters that we're going to pass. This is data that we're going to pass to the function. Those arguments or parameters are going to be stored in the stack frame.
- And we also store control information. So remember when you call the function, you are doing control transfer, from where you were, to where this function code is. When you get done with the function, you have to return to the point from where you had made the call.

A Vulnerable Password Checking Program



```
#include <stdio.h>
#include <strings.h>

int main(int argc, char *argv[]) {
    int allow_login = 0;
    char pwdstr[12];
    char targetpwd[12] = "MyPwd123";
    gets(pwdstr);
    if (strncmp(pwdstr, targetpwd, 12) == 0)
        allow_login = 1;

    if (allow_login == 0)
        printf("Login request rejected");
    else
        printf("Login request allowed");
}
```

This program is vulnerable. This really simplified program what it's doing is something similar to what a password checking program might do.

Think about what the program is doing, this is our main function as the program. It takes some arguments. It has bunch of local variables. It has an integer that's value at the end is going to be should login be allowed or not allowed. It has a local variable of size no more than 12. Well that's where

we're going to read the password that we're going to ask the user to type. And then we have to compare that with something that we know about the password of the user. So the target password here really is the password we are looking for. Here is a really poor practice here, but the target password is hard coded. Other thing I should mention here is that password checking programs actually don't need to keep your password because then if somebody gains control of that then they know your password also.

So these are sort of the local variables that we declare in this function. This is where we get to this code. The code is very simple. It's saying I'm going to get a string. Gets is read a string function get a string that you're going to type. We're going to read that string into this local variable `pwdstr` that we declared, then we're going to do a comparison of the string that we just read into `pwdstr` against our target password, which is "MyPwd123". This string compare actually is, this variant says at most 12 character long strings and the result of this comparison, if the two strings are identical then it returns a 0, otherwise it returns a non-zero value. So read more about this function to find out exactly what it returns in the other cases, less than zero or greater than zero. So in this case, "allow_login = 1", after that we going to say if "allow_login = 0", remember that was the initial value, we had shown that we don't allow unless there is a match. Remember the failsafe defaults we talked about? So the default here is don't allow. So that's why we're setting it to zero. So if the variable is not set to 1 as a result of a successful comparison then it's still 0 then in that case we are going to reject the log in request. Otherwise the match was successful. We set it to 1, so in that case, we're going to allow the log in request. But the goal here is to understand how vulnerable our code gets exploited.

So we want to see how this code gets exploited. And this simple program that we have here helps us do that. And the vulnerability that we're going to explore today, which is buffer overflow, stack buffer overflow is actually going to manipulate the memory that is in this frame. Before we actually get into the details, I should mention the software vulnerabilities that we have. An attacker is allowed to call a program that you run on your system. The program may have an interface that allows legitimate users

to make calls to it. And when they make those calls they are actually going to pass some data, so the entry point here is actually a legitimate call that could be made by users of this program. They pass certain kind of data and the attacker is going to pass data that is not what we expect. As a result it's going to lead to this overflow that we're talking about.



Stack Access Quiz

Check the lines of code, when executed, accesses addresses in the **stack frame for main()**:

```
int main(int argc, char *argv[]) {
    int allow_login = 0;
    char pwdstr[12];
☐ char targetpwd[12] = "MyPwd123";
☐ gets(pwdstr);
☐ if (strcmp(pwdstr, targetpwd, 12) == 0)
        ☐ allow_login = 1;

☐ if (allow_login == 0)
        ☐ printf("Login request rejected");
☐ else
        ☐ printf("Login request allowed");
}
```

```
int main(int argc, char *argv[]) {
    int allow_login = 0;
    char pwdstr[12];
☒ char targetpwd[12] = "MyPwd123";
☒ gets(pwdstr);
☒ if (strcmp(pwdstr, targetpwd, 12) == 0)
        ☒ allow_login = 1;

☒ if (allow_login == 0)
        ☐ printf("Login request rejected");
☐ else
        ☐ printf("Login request allowed");
}
```

This quiz is actually asking you to go through this code, and see exactly what it's doing. So, where is that data? Is it on the stack in the stack frame that we just talked about. If the answer is yes, then you check it. If the answer is no, then you don't check it. So you go through each line. Understand what the code in that line is doing, and what data it's accessing. If you think it's accessing data on the stack, then you check it.

So, let's look at what's happening here in this declaration.

We're saying not only give us a variable called targetpwd that is 12 characters long, but assign the value MyPwd123 to it. So initialize this variable with this value that we are providing. So we have to actually store this MyPwd123 into this variable that we are allocating in the stack. So this are the initialized we're going to access it.

What does gets do? I have checked that as well. So gets remember, reads a string, that's user input. And stores that into this variable called pwdstr. So remember we declared

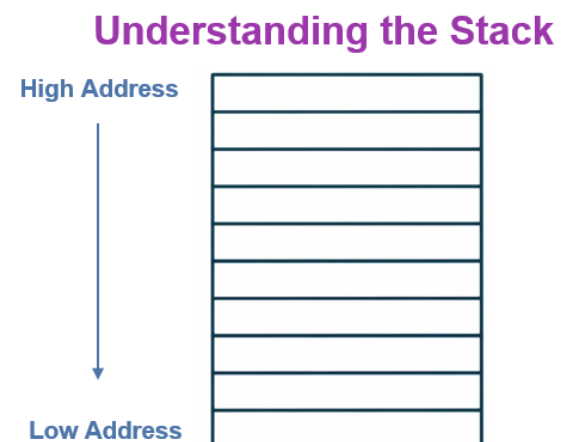
again at size 12, this character string that's where we're reading. So when we gets is going to ask for input, it's going to take that input and place that input in this pwdstr. As a result, it needs to access this variable and we know this variable is allocated on the stack. So, in particular the stack frame for this function call, so it's going to access addresses when we do gets as well.

Next statement, you see several of them checked because they're all access variables that are being allocated on the stack. That's what we're trying to illustrate here. So here, this is comparing two strings. Again, the string strcmp is string compare and tells you the size of the string, the max size of the string. So it saying compare password string that we just was input. Target password that has this MyPwd123, so to compare these what do we have to do? We have to go fetch those values. Character by character or whatever this scheme is that we're using to do the comparison. But we have to see what is in this variable. And what is stored here so to read those values again, we have to access the stack because remember both the the local variables are allocated in the stack frame.

Once we do the comparison, if the comparison is successful, we are writing this variable. We are writing one into this allow_login variable. Storing this value into this variable that's on the stack that was allocated in the stack frame is again going to require that we access the stack. And as a result, we have checked it because this again requires access to this variable that's on the stack.

In this if statement that we have here, we have to find out what is the current content of this variable `allow_login`. So we have to read it. Earlier we stored a value in it. If the comparison was not successful, actually we will not do that. So this will not be executed if the two strings didn't match, but in case they did match, we store one. But when we come here we're saying, we're going to check this variable `allow_login`. If its value is zero, then we're going to print one message, if it's not then we're going to print a different message. So to find out the value of `allow_login` we have to read it which requires access to an address that is in the stack frame.

Then we're doing this printing, so if you look at these are function calls. And I told you earlier function calls result in a stack frame when you make the call. But remember they're not accessing any variables that we have in our current frame. So, I haven't checked these three boxes here because we are not accessing any of the variables that exists over that allocated space in the current stack frame. That is what we are discussing here for this particular function.



Remember I said, processes or programs execute an address space. It's a linear address space. It goes from zero to some maximum address, depending on what your address space size is. And a little bit more complicated, we'll get to that when we talk about operating systems. Part of the address space is where the operating system goes, but the other part is where user code and data is going to go.

So that's the part we're talking about. There's some place where the code is going to go, that's called segment, that long lived data goes into part of that other space that's called the heap. And the temporary

or dynamically allocated data that we were just talking about, when you make function calls, goes into the stack. That's what the stack is used for.

So we know a stack is, basically the two things you can do with the stack is push things on it. So there's a stack pointer. When you push something that is put on top of the stack and the stack pointer moves one place. The other thing you can do with a stack is pop something off it. So when you do a pop operation, then the data item or the element that is at the top of the stack, the stack pointer it's pointing to, actually gets popped off, put in some other place, a register or a memory location different from the stack. And then the stack pointer is adjusted to reflect the fact that that item is no longer there.

So if you sort of think about the stack as you do push and pop, the stack pointer moves. And you need to sort of see in what direction the stack grows. So when you push things onto the stack, the stack is growing. You're adding more items to it. When you pop things off, the stack is shrinking, and the stack pointer moves as the stack grows or shrinks. So we are going to assume that the stack actually grows from high addresses to low addresses. So the stack pointer starts at some address. And remember that address space starts at zero and goes to the maximum so we have this increasing addresses, address n , $n + 1$, $n + 2$. Typically it is byte addressable, so each unit for which we have an address is a byte, or eight bits. So the addresses increase as we go down the address space. So when I say the stack is going from high addresses to low addresses, what does that mean? Well, the stack pointer actually absolute address value, it's going to decrease as we push things onto the stack. We allocated some amount of

room for the stack, or some amount of space for the stack and grow and shrink. And the stack pointer is initialized to the bottom of that space. So the highest address that we have for the stack reason. And as we push things onto the stack, the stack moves to a lower address.



Attacker Bad Input Quiz

What **type of password string** could defeat the **password check code**? (Check all that apply)

```
#include <stdio.h>
#include <strings.h>
```

```
int main(int argc, char *argv[]) {
    int allow_login = 0;
    char pwdstr[12];
    char targetpwd[12] = "MyPwd123";
    gets(pwdstr);
    if (strncmp(pwdstr, targetpwd, 12) == 0)
        allow_login = 1;

    if (allow_login == 0)
        printf("Login request rejected");
    else
        printf("Login request allowed");
}
```

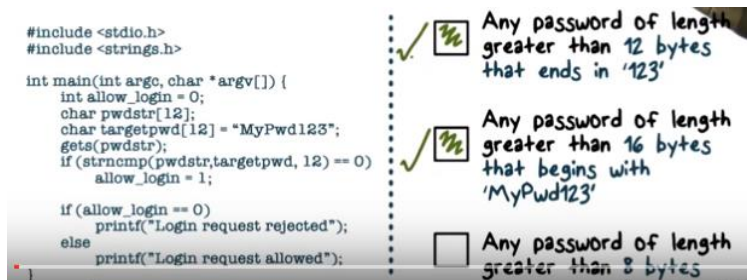
- ☐ Any password of length greater than **12 bytes that ends in '123'**
- ☐ Any password of length greater than **16 bytes that begins with 'MyPwd123'**
- ☐ Any password of length greater than **8 bytes**

show you how exactly how we end up exploiting it, I want you to sort of think about it. So I want you to sort of look at these three possibilities we have. The input that I pass is greater than 12 bytes and ends in 123. Any password of length greater than 16 bytes begins with MyPwd123, which is the correct password. Or any password of length greater than 8 bytes. Any string, only requirement we have here is that it should be more than 8 bytes.

So what would happen if we passed input to this program? Extremely important, the vulnerability in the exploit is going to rely on that input. So I'm asking you, think about the three different inputs that we have here in these three choices. And, could defeat, the answer to that is that whatever check that we're providing, that should fail. Okay, defeating here just means the password check should fail. So remember, if we exactly pass this the answer should be yes. If it's anything different than this, the answer should be no. And rejected or allowed should be printed at the end. So defeating means actually producing a result that is not what I just said. Giving incorrect password, and getting an allowed answer, that would be bad. That way you defeat the check. Or giving the right password and getting rejected is also defeating the check that we're talking about. So think about these three inputs. Think about what I just said about, what does it mean to defeat the check that we're doing here. And check all that result in the check getting manipulated or being defeated as we say here.

At this point, we have seen the code that we are examining for a vulnerability and we'll soon discover that. We talked about the stack where some of these variables that we have in this code are going to be allocated space.

So, to further sort of reinforce what our goal here is to exploit that code, or exploit that program, we're going to do another quiz. And before I



So the first one I've checked here that it could defeat is any password of length greater than 12 bytes. So then what happens when give it a password that is greater than 12 bytes? This gets that we have here, is going to read whatever string you provide to it. The one we have here is only for 12

characters. So it's 12 bytes long. But when we executing this function says, get string. It has no idea. It's going to accept a string. Whatever length you provide. Since it's greater than 12 bytes. The first 12 characters is going to get stored here. And those are going to be compared with this. And where would the remaining characters go? Well remember this variable starts some place in the stack frame that we are talking about. On the stack, it's allocated space. And when the 12 bytes run out, and we have more input, what do we do? We keep going. So, starting address, and then we fill the memory starting at that address, as with the data that we have. So, if it's 20 characters, we're going to store these characters we get in the string in the next 20 bytes. So think about what happens if this variable allowed login is right after this variable password string. If that's the case, then the bytes of password string, that are beyond 12 okay, the characters in these string that come after the first 12, actually going to go to memory we have assigned to this variable. Initial value is zero, this characters that we're going to read and store in the locations or memory that is actually allowed for that's allocated for allowed login. Well, that value zero is going to change to something else depending on whatever character that we're reading. Even if the string doesn't match here. Because this is whatever string we are saying, any password so it doesn't match. We are not going to assign one here. We're not going to come and sign allow login one here. After the comparison that we do, but some data has already overflowed when we read this strength into this variable. And because that's what's going to happen when we check, do this check here to this side if login could be rejected or allowed. Is it going to be zero? It won't be zero because some data. That was input as a result of gets function that we called, has written into this location that was allocated, so we have this overflow.

Remember, we started talking about memory overflow, or buffer overflow. So, if you sort of assume that there was some likelihood that this variable was sitting right next to this variable. And if that is the case, then if we had too much data for password string it will end getting stored into this variable that will change its value from zero to something else, and the outcome of this check will be very different. So even if the password is different from my password one, two, three, and this check fails and we don't set it here. This is still going to say login allowed, because it's going to find a known zero value. Because this zero value has been corrupted because of the overflow that we have. Because the string that we read was greater than 12 bytes. So it keeps going, writes into the space that belongs to this variable and as a result its value is no longer zero. Its value is no longer what it initially was, which was in the case zero.

So this is the way to think about how we defeated it by changing the value of a variable that is important obviously for making, deciding which way this check goes. Corrupting it by supplying input. Okay, that exceeds the amount we had allocated for this variable in which we were reading this input. So all our flows always going to be something like this. It's going to all float, over write something, and we going to talk about read only book flows also. But typically, it's going to write and corrupt those

values or going to write some values that attacker wants. So in this case, that hacker wants log in to be allowed so, doesn't have to guess the password. By carefully crafting input and over writing this variable that hacker is able to achieve the outcome, which in this case is going to be able to log in by doing this.

The same thing would happen in case two because we haven't all four here as well. So any password of length greater than 16 bytes, it doesn't matter where it begins with. Lets say after that because we had checking 12 bytes, we going to check the eight that we have here. So the next set of characters may be different than what the initialized value here was. So even if there isn't a match. There is an overflow and because of that we have the same stuff happened that happened before. Again, our input is not good here. It is larger than the expected input which is no more than 12, so that's why we can potentially defeat it. And we'll see exactly how that is happening.

If the password is greater than eight bytes. Well as long as it's not. So it's any password. It doesn't have to start with my or my password or whatever it is. But if it's nine bytes ten bytes 11 bytes. As you keep going. Until the amount of space that we have here. This will not fail. Because we're not overflowing. But if it's greater than 12 as we saw before, then we get into the same. So if the password was nine characters long, the check will not fail. And that's why I did not check this. The password check code is going to output the right value because we don't have an overflow.

So, here we talked about playing with or manipulating the logic of this program, which is the password checking, and achieving that or accomplishing that by providing input that is bad. We considered some examples of input where the password is different than this password that was expected for a certain user, but we still able to get this program to say that our login should be allowed. Our login request is proper and should be allowed. So this is what bad input can do to a program. We're not changing the instructions. Before we call the function, we didn't get our hands on this code and modify, okay, that's a lot harder to do. We just calling a function that we're allowed to do, and give it a password, except we're giving it a bad input when it ask for a password and that's how we're manipulating the execution of this code.

Attacker Code Execution

We type a **correct password (MyPwd123)** of less than 12 characters:



The login request is allowed.

Now let us type "**BadPassWd**" when we are asked to provide the password:



The login request is rejected.

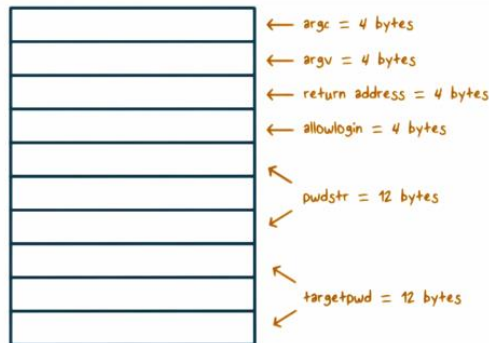
Let's just look at this code execution that we're talking about. If the attacker actually guesses our correct password, and types that as input to the program, login is going to be allowed.

What they're saying here is the attacker doesn't have to do that, doesn't have to guess the password, the correct password. So it's guessing some password, might takes trying different types of passwords and so on. It gives a BadPassWd and we don't have an over

flow. Because the BadPassWd here, if you look at it, it's still three, four, seven, nine, ten characters long terminated by a null. We still are good, because we don't have an overflow. And when we don't have an overflow, this actually fits into the space we have for the password string. So, no overflow, and it doesn't

match, so as a result the request is going to be rejected. So this is sort of the things you do. You either find a password, then you have access. You can become that user and do what that user is allowed to do. Or you can try a password, but you're not going to be successful by just trying any password. The chances are, the password you type is going to be, is not going to match the user's password. And match fails, login fails.

Attacker Code Execution



So we looked at that program, what it does, what input it takes, how does it execute on the different values for the input that we provide it. But to really understand how we actually take this buffer overflow to its logical conclusion, which is actually getting control of this program. We need to actually look carefully at how data is stored on the stack. What is the layout of the stack?

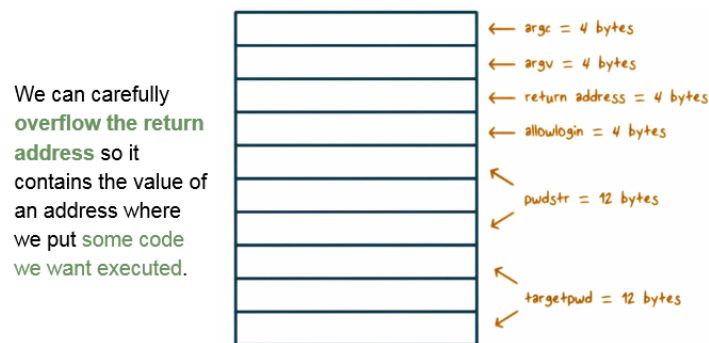
So remember I told you the stack grows from high addresses to low addresses. So we are at some address, and as you put more stuff on it, we are going

to lower addresses. So as we go down the stack, as we push things on to the stack, we get to lower and lower addresses and that's what I meant by the stack growing from high addresses to low addresses or growing up few. Think of your address base starting at zero and going to the max. Then the stack growth is from higher to lower addresses. And lower addresses are higher or up. So that's why we're saying it grows in that direction.

So when you make the function call, we're going to push the arguments `argc`, with the arguments. So we are assuming this is what the compiler does. You make a function call in your programming language. First thing it does it's going to push. So wherever stack pointer is pointing, we push this argument that takes 4 bytes. So the place it points to the address where the stack pointer is pointing is address 4 `addr - 4`. Then we push `argv` and at this point we have to push the return address. Remember we have to know where the return address is, so when we finish executing this function we can go back to where we came from. You then push the return address, so the compiler generates code to push the arguments. And then you make a function call or you make a call. And typically the hardware pushes the return address onto the stack wherever the stack pointer is pointing. That goes and then we get to the stack pointer and we decrement the stack pointer by 4 because the return address typically takes 4 bytes [32 bit OS]. So now we're going to allocate space for the local variables, `allow login` takes four bytes. 12 bytes strings take 12 bytes that we have here. So each of password string and target password we have 12 bytes each. So, you made the call, if the stack pointer was pointing to `Addr`, then I can say, well the address of `allowed login` is `Addr-12` because 4 for `argc`, 4 for `argv`, 4 for return address. Already occupied these 12 locations, so the stack pointer has to move beyond those. And it does that by moving 12 bytes up and as it goes up to lower addresses, we're doing the subtraction that I said.

So, now if you look here there's a couple of interesting things. This is password string isn't it? It's allocated space starting here. If it's less than 12 bytes, the input we provide, it's going to stay within these 12 bytes that we allocated for it. So this is the starting address, so if the variable is that address, address +1, +2, it goes like that every byte you store. So these are the higher addresses we were talking about. So, password strings start here, if it's less than 12 bytes, it's only going to go into these 12 bytes. What happens if it's longer than 12 bytes? Some of the examples that we did when we set it to 16 or something like that, it's going to exhaust these 12 bytes. It's going to overwrite the allowlogin and if it's more than 16 what is it going to do? It's going to overwrite the return address. So the way these things are let out on the stack, remember our password string variable address starts here. And as we provide some number of characters it starts storing them. If they're less than 12 we're good. If they're more than 12 we're first going to overwrite allow login. Then we're going to overwrite return address. Then we're going to corrupt the argument values and things like that. Now we can say well, what if I give more than 12 bytes. We earlier said 16 or ending whatever it is. We know that if it's more than 12, it's going to all flow into this. This is what it is going to over flow is.

Attacker Code Execution



Depending on what sort of input data we provide when this string is read, we know that if we provide more than 12 bytes, we're going to overflow in this direction into these variables as we're talking about. So what can you do if you, sort of, are careful about it?

First thing that we want to do is, we want program control to go someplace where our code, the attacker can actually craft some code. There are two things the

attacker has to do. It has to have its code that it wants to run, and then it has to find a way to get the program control transfer to happen, such that the program goes and starts executing the code that the attacker has crafted.

Here, we should sort of look at this and say there is an opportunity here for the attacker. The attacker has found a way to go up and write into this locations. We know that it has done that, because if it passes more than 12 bytes these 4 bytes are going to be overwritten, that's allowed login. If it writes more than 16 bytes, then the return address is gonna be.

And so if we carefully sort of overflow, and address where our code is. And the overflow actually ends up writing the address of that code into this return address. What would happen in that case? The function, when it's done, will not return to where it came from. Remember it had stored the address where it needs go back in return address. But by carefully sort of overflowing and modifying this return address, that old return address is history. It's gone. It's going to actually return to the address that we have inserted or we have put in these 4 bytes. The 4 bytes where we have the new address that we have put in to this area of memory where we have the return address before. That's where this program is going to return.

So this is kind of an ha moment. By providing bad input, the vulnerability is, we're going to know that, we're not checking input. That's the vulnerability. That is what is going exploited by that hacker. The

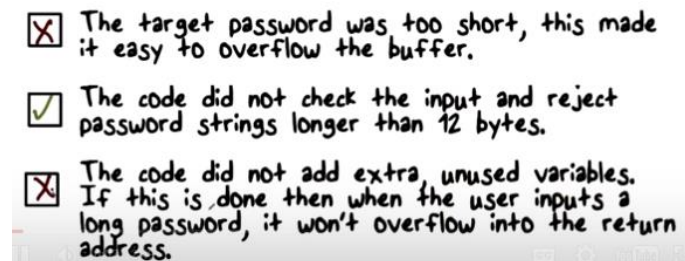
attacker is providing a carefully crafted input. And when we take that input, an overflow occurs. And the way the overflow reaches the return address area, the part of memory where the return address is stored. We are actually going to put a new return address where code that we have created, by we I mean the attacker, the program will start executing that code. So some code that hacker wants executed, if we know where that code is, and we know its address, and this is what we override these four locations with, the return address by passing bad input, exploiting the vulnerability, when the program doesn't check it's input. By passing carefully crafted input, we are now able to take the program from where it was to a place where our instructions are located.



Buffer Overflow Quiz

Which of these **vulnerabilities** applies to the code:

- ☐ The target password was too short, this made it easy to overflow the buffer.
- ☐ The code did not check the input and reject password strings longer than 12 bytes.
- ☐ The code did not add extra, unused variables. If this is done then when the user inputs a long password, it won't overflow into the return address.



So let's talk about a quiz that's sort of revisits this idea of a vulnerability and resulting in this exploit that we are talking about. So we know that there is a vulnerability and what is the reason for that vulnerability? That's what we are trying to answer in this quiz.

Okay let's talk about the answers. Remember the quiz was about what is the source. What did you or did not you do when tried to write the code for that password checking program that resulted in a vulnerability?

So, the target password was too short, this made it easy to overflow the buffer. That's

not the vulnerability, isn't it? It doesn't matter how long the password is going to be. Maybe you can expect people to have passwords that are 100 characters long. But if you assumed the password was not going to be more than 100 characters, and someone gave it 110 characters, the overflow would still happen. So it's not the absolute length of the password that is the issue. The issue is that we allocate a certain amount of space for the maximum, and the maximum could be anything. So, it being too short is not the issue.

The code did not check the input and reject. This actually is the source of the vulnerability. Later we're going to remind you that secure coding, golden rule is check your inputs. We didn't check our inputs. We made an assumption that the input will never be more than 12 bytes. Because that's how much space we allocated for the password string variable. That's where we're going to store the input the user provides. We made that assumption, what could have we done beyond that? We could have, when we received the input, actually checked it. If the input string is more than 12 bytes that's an overflow. And we should have rejected and not gone on and done the comparison and things like that. So the code did not check input and reject password strings that are longer than 12. That is the vulnerability. Correct

answer here is, the vulnerability arises because the input is not checked. And when the input is not checked that results in overflow.

Over the last answer, the last answer says the code did not add extra, unused variables. First of all, there's no reason why you should add dummy variables, but even if you did by making that input string longer and longer, we can always sort of go past those local with the additional variables. So, just adding additional variables. Now, create some distance between the return address, and where the password string variable is stored. But, by giving longer input, we can go past those. So that is not, again, a correct answer, so the vulnerability here really is in that code that we have.

It's poor programming practice, a lot of people unfortunately make that kind of mistake, is the code did not check its input and because of that there was an overflow. And overflow running into the return address. And that gives the attacker a way to direct program control to some other place where attacker crafted code may be available.

ShellCode

Shell Code: creates a shell which allows it to execute any code the attacker wants.

Whose **privileges** are used when attacker code is executed?

- The host program's
- System service or OS root privileges



LEAST Privilege is IMPORTANT

can ask that that particular code be executed. So let's talk a little bit about how do you craft the Shell Code. Say call you're making to launch a shell, you can write the code in C for example. It's a fairly short piece of code, you're just using one of the calls. And then let's say you compile it into assembler instructions. So once you have assembly instructions, remember the shell code has to be machine code. Because these are values and they have to be encoded properly and things like that. These are values that you are going to store in memory. That is where your shell code is. That's where control is going to be transferred. So memory is going to store these binary values. So it may look like data, but actually it really represents instructions of codes. And if your program is instructed to go execute from there, then it is going to be a program. So you really have to figure out what the instructions are and what data those instructions are going to use when they execute. And you're going to craft that as a set of values that you load into this part of memory. And that's going to be based on the instructional codes of the machine code as I've said before. But the way you can do that is by writing the code in a high level language, Assembler, translating that to machine code and that's how you figure out what is the ShellCode that you're going to need. So Shell Code that we going to place somewhere in memory, this is how you're going to get to it. Then I say, we know how to transfer control to it, because we modify the return address when the function ended, it will actually go to this place where the shell code is.

So, let's ask ourselves this question. When control transfers to the ShellCode, what privileges are going to be used when this code is executed? We know the code is attacker code. The attacker actually created the code, wasted memory, transferred control of this program that we were running before to

So the code that the attacker wants to craft is basically code that is going to launch this command shell that we're talking about. And that kind of code is called Shell Code.

So how do you write Shell Code? Well a little bit of expertise comes in. So the Shell Code as I said creates a shell, which is going to allow you to execute arbitrary commands. So any code program that we have, the attacker

the ShellCode, and remember the program was running on behalf of some users. So it's a service, it may be running on behalf of rule or depending on what sort of rule is this or credentials we have some role or whatever it is. It could a system process for example. So, the program was running with certain privileges. This program is now going and executing these instructions that make up the ShellCode.

So whose privileges are we going to be using when the shell code or the instructions that make up a ShellCode get executed? These privileges are going to be the privileges of the host program, the program that gets exploited was running with certain privileges. This attacker code now has the same privileges that the host program or the target program that could exploit it had. So if this happened to be system service as I said, is running with root privileges. Essentially, you have keys to the kingdom. You can access arbitrary resources, so the attacker used a vulnerable program used above for all the idea that we talked about, transfer the control to its own code, that was the ShellCode, and the ShellCode lets it launch to command shell, and the command shell now can start whatever program it wants to and that program would run on behalf of the host program's owner or group privileges whatever that it had.

So the attacker has come in and become you, either the system or a particular user who is running this vulnerable program. So, this is how the attacker is able to execute arbitrary code with a legitimate user's credentials or privileges. Legitimate user's privileges are available to this arbitrary code that is being executed on behalf of the attacker. That's the best case scenario from the attacker's point of view. It's the worst case scenario from our point of view because somebody is executing code we have no knowledge of with our privileges. And they were able to get into the system because there was a vulnerability that got successfully exploited and as a result the attacker has ability to do everything that he or she wants to do.



National Vulnerability Database (NVD) Quiz

- How many CVE (Common Vulnerability and Exposure) vulnerabilities do you think NVD will have?

[1] Close to 500, [2] A few thousand, [3] Close to 70000

- If you search the NVD, how many buffer overflow vulnerabilities will be reported from the last three months?

[1] less than 10, [2] Several hundred, [3] Close to one hundred

- How many buffer overflow vulnerabilities in the last 3 years?

[1] Over a thousand, [2] fifty thousand, [3] five hundred

So this quiz is about how common are these vulnerabilities, or how many vulnerabilities are known. So remember that vulnerabilities we don't know about. Okay we come to know about them only after we are exploited. And those are called zero day vulnerabilities. We had no time to fix them or patch them before they got exploited. But known vulnerabilities, you patch your systems to make them go away in things like that. So we're talking

about what systems actually have known vulnerabilities. So the data about this is maintain this National Vulnerability Database or NVD [<https://nvd.nist.gov/>], so you will need to access that to answer this quiz. And it has lot of Information about vulnerabilities that are numbered through what is called CVEs and it tells you what system nature of the vulnerability what can happen and things like that. But here rather than go into those kind of details we're just talking about how many of these are there. So take a few minutes to think about these questions and take answers and then we'll come back and discuss what the correct answers are.



National Vulnerability Database (NVD) Quiz

- How many CVE (Common Vulnerability and Exposure) vulnerabilities do you think NVD will have?
[1] Close to 500, [2] A few thousand, [3] Close to 70000
- If you search the NVD, how many buffer overflow vulnerabilities will be reported from the last three months?
[1] less than 10, [2] Several hundred, [3] Close to one hundred
- How many buffer overflow vulnerabilities in the last 3 years?
[1] Over a thousand, [2] fifty thousand, [3] five hundred

Let's talk about answers to these questions we have, about how many vulnerabilities do we know about in software that is running on our systems.

So, first there's how many vulnerabilities do you think are there in the NVD? Unfortunately, the number is the largest possible that we have is the answers is close to 70,000. The last time I checked it was 69,000

something. We're talking about tens of thousands of vulnerability. To that there is millions and millions of lines code out there that have been developed by all kind of people and companies and things like that. But these vulnerabilities that can be exploited by attackers are not necessarily very rare, we know about lots and lots of them so that's answer is 70,000. So then we going to focus just on buffer overflows, we know what exactly buffer overflows are. We going to do our, fix our vulnerability for checking or things like that. We know we can make them go away.

Well, the last three months if you look at, and this is sometime in middle of March in 2015, we actually have close to 100, actually the number was 107 I think, buffer overflow vulnerabilities that have been reported to the National Vulnerability Database in just the last three months. If you look at the last three years, the number is about 1,000, a little over 1,000, which means the rate at which these come at us is not necessarily decreasing. Because in three months, if you have close to 100, in a year you'll have about 400. In three years we'll have 1,200, so it's in the range. Looks like we're seeing this vulnerabilities becoming known at the rate that doesn't seem to be going down significantly, compared to two years ago or something like that. So the answers are that even when we focus on buffer overflow, we are seeing almost a vulnerability every day; Close to 100 over three months - that's about 90-some days. So we're seeing almost a new buffer overflow vulnerability every day, leading to thousands of those or close to 1,000 in the last three years.

So the takeaway here is that software vulnerabilities. Now we know how they get exploited. They are out there and they provide a very viable path for attackers to craft attacks and gain control of our systems. And we know that, we discovering them almost on a daily basis. And the attackers perhaps know about them. They discover them before you do, and then they're able to exploit them. Because you don't have any defenses or any widely deployed patches to fix them.

Variations of Buffer Overflow

- **Return-to-libc:** the return address is overwritten to point to a standard library function.
- **Heap Overflows:** data stored in the heap is overwritten. Data can be tables of function pointers.
- **OpenSSL Heartbleed Vulnerability:** read much more of the buffer than just the data, which may include sensitive data.

So far we have talked about stack buffer overflows. Remember we started with the idea of both overflows and then I quickly narrowed onto stack buffer overflows or overflows on the stack. There are other variations of buffer overflows and the number of reasons for those. We want to talk about these different types or variations of buffer overflows before we actually start

talking works and defenses against these kinds of attacks that exploit buffer overflow.

So the first variation we're going to talk about is what is called return-to-libc. libc is the library, C library, of a program's use library functions. So, remember when I was talking about buffer overflow shell code, modifying return address. Our assumption was that we're going to return address, it's going to be modified to point to point to some place where we are able to place the shell code. And that could be on the stack itself. In that case you were transfer control to the shell code and execute it off the stack because that's where the code is stored, the shell code is stored. We're going to talk about some defenses where systems don't allow you to do that. So it's not easy to find room for your shell code on the stack and get it executed. So the variations of where you should return, you don't have to return to code that is explicitly write as shell code, but you can return to a function in the library. So the return address is going to be modified. The return address that we had on the stack, it's going to be modified to point to a standard library function. So the assumption here is that you'll be able to figure out the address of the library function. And once you have that, that's what is going to overwrite. So when you do the stack buffer overflow you make sure that library function address is what gets returned into the return address field on the stack frame.

So why do we want to do that? Why do you want to return to a library function? Well if you return to the right kind of library function. And you're able to setup the arguments for it. Or the parameters before the call happens on the stack. Then you can execute a library function with arguments or parameters of your choice so think about the library function called system. The system call, if you look at it carefully, you can ask it to execute bin csh or some csh or something like that. And as a result of executing this function call you would launch a command shell. That's what our shell code was doing before. Now we can get the same result by executing a library function. And the reason it will do that for us is that before we make the call to it we go to it by returning from this function where we were able to overflow the buffer. We're going to set the stack in such a way that the arguments actually are going to be such. So for example I said the system calls should execute bin something. It's going to be such that when hit the system library function or the library function's system, it's actually going to do what you want it do. In particular give you a shell command. A command shell. So when that happens, you actually then going to execute code that you didn't have to craft and place on the stack. You're just executing a library function. And you're manipulating its input by properly crafting that input on the stack. And before you make this library function call. And then it's going to do your bidding if you are that hacker.

So in libc, the thing to remember is that the return address is modified to point to a chosen library function and the setup is input in such a way that the execution library function, with that input, allows the attacker to sort of gain control the same way we were able to do before with shell code.

- **Heap Overflows:** data stored in the heap is overwritten.
Data can be tables of function pointers.

Instructor Notes

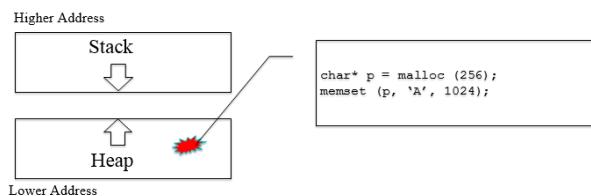
[Android Lock Screen Attack](#)

Another variation is that overflow occurs. So, by overflow we mean, we write it to beyond the point that this variable getting written, has been allocated space. So, I said data also gets allocated on the heap in a program. So long-lived data, for example, global variables, and so on, get stored on the heap. So, one crucial difference between the heap and stack is that heap does not have a return address. So, you cannot hijack the control flow of the program, and take it some place where you have your own codes, or a library function, or return-to-lib kind of attacks. But in the heap we do have function pointers depending on what kind of language, and how they implemented and so on. So data can be these tables of function pointers. And there you can modify a function pointer, and by doing that you can transfer control, it's actually a lot harder than the stack buffer overflows we were are talking about. Heap overflows require sort of more sophisticated in some sense and require more work, but you can by modifying, figuring out where a certain function is, and modifying the function pointer, you can transfer control to somewhere else where this new address that you place in the function pointer points to. So heap overflow is, again, we're corrupting memory in some sense as we did before, but this happens to be memory that is in the heap part of the address space.

●Heap Overflow

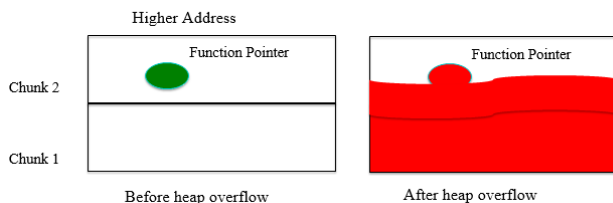
- Buffer overflows that occur in the heap data area.

- Typical heap manipulation functions: malloc()/free()



●Heap Overflow – Example

- Overwrite the function pointer in the adjacent buffer



- **OpenSSL Heartbleed Vulnerability:** read much more of the buffer than just the data, which may include sensitive data.

Now, so far, any time I talk about overflow, I indicated that we're writing. That we're storing data in some part of memory, and we keep going beyond the limit where we should stop. And so, the overflow is the result of writing stuff. Overflows don't just have to be associated with writing data. Overflows could also happen because we read. The whole idea here is that overflow is because we do too much of something. We keep writing beyond the limit, so we write more than what we're supposed to.

What if you read too much? Let's say a variable has 12 bytes but you asked to read 100 bytes. What the read is going to do is it's going to keep going beyond the variable and it's going to go on to the next one and the one after that, and things like that. So wherever in memory you have that variable that you're reading, if you go beyond that, you're going to get into other variables.

Now, there is a fairly well-known vulnerability in the open SSL code which is used to secure all kinds of online transactions, and secure communications, and so on. It was called the Heartbleed Vulnerability.

Actually, it was an overflow related vulnerability. But it wasn't writing into the stack beyond the memory that we had allocated for the variable into which we are writing. It was actually a read overflow vulnerability. Actually, this OpenSSL Heartbleed did, is that it kept reading beyond the variable that you're supposed to read. And there was some juicy stuff beyond that to do with keys and things like that. So if you read too much, you're going to get additional data beyond what's really should be in this variable that you're trying to read. And so, the additional data that you got could be used to figure out sensitive keys and things like that, and that's what the vulnerability was. But the vulnerability comes from the fact that we're reading beyond the variable that the code was supposed to access. As a result, the overflow occurs while you are reading. You go beyond the boundary of the variable and keep going. So that's called a Read-only Buffer Overflow, and the example that we just discussed is the Heartbleed Vulnerability that occurred because of that.

Buffer overflows don't have to be on attack only. They can happen on the heap, and buffer overflows don't have to be just when you write, they can also happen when you read. It's just going beyond the boundary where you should stop. You don't do that, that's the overflow. If you're reading some data, then it's read-only buffer overflow.

Defense Against Buffer Overflow Attacks

Programming language choice is crucial.

The language...

- Should be **strongly typed**
- Should do **automatic bounds checks**
- Should do **automatic memory management**



Examples of **Safe languages**: Java, C++, Python

good thing. We shouldn't write code that has those vulnerabilities, but if such code is out there deployed on systems, we need to find ways to defend against attacks that are going to exploit these kind of vulnerabilities. So we're going to talk about a number of defenses that help us counter attacks that rely on these buffer overflows.

The first one we're going to talk about is your choice of programming language. What language are you writing your program in? There are actually going to be languages where this kind of problem goes away, you don't have buffer overflows. And these languages are languages that are strongly typed, that do lot of memory management they do automatic automatically, so they do bounds check. Remember one of the problems that we had was that we declared a string variable of size 12 and we said you can read any amount into it. There is no way unless I explicitly inserted a check, the language didn't do that, unless I explicitly inserted a check that said giving it more than 12 bites, that's not good. The language didn't stop me from clobbering whatever was next to the password string that we had. Not all languages do that. There are languages that say you have an object of size this, this is how much it could be, it can't be any more. So what type of variable it is, how much memory is required for it or is allocated to it, and how much memory you can access when you access that variable, this bounds check that we're talking

Instructor Notes

[Buffer Overflow Explained](#)
[Buffer Overflows Explained](#)
[Source Code Analysis Tools](#)

So we have talked about buffer overflows and how programs that have this vulnerability can get exploited, in particular letting attackers or giving them the ability to execute arbitrary code with the host services privileges. It's not a

about. And if you manage this memory automatically, the problem of memory overflow, or abusing memory the way we did when we have buffer overflows that goes away.

So, languages that do are called safe languages, because you can rely on type safety and you don't make these kind of mistakes. So there are several examples of these languages, Java, C++, and others. So if you write code in these kind of languages that have type safety and that do the kind of things that we're talking about, buffer overflow would not be a problem. The buffer overflow typically comes because we don't do either strong type checking or this bounds checks that I was talking about, which is what low level kind of languages are notorious for that. But these object oriented strongly typed languages make that problem go away.

Defense Against Buffer Overflow Attacks



Why are some languages safe?

- Buffer overflow becomes impossible due to runtime system checks

The drawback of secure languages

- Possible performance degradation

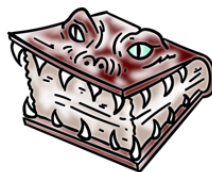
If you go for this defense where you use languages that are safe, obviously buffer overflow becomes impossible because of all the checks the language is going to do, so we set bounds checks. So the checks have to be done at run time rather than you doing that explicitly in the code that you write, that language is doing for you. So, why don't we use

these kind of languages for everything? Well more and more we're doing that but one drawback with these kind of languages typically is that, that's going to be either some sort of performance degradation because we just talked about the checks that had to be done. And they force you to do certain things in a certain way. So flexibility that generally you have you have with low level programming languages goes away. So if you're looking for really performance or all the flexibility in the world, maybe that's why you don't use these languages that do this work for you to make the buffer overflow kind of problems go away.

Defense Against Buffer Overflow Attacks

When Using Unsafe Languages:

- Check input (**ALL input is EVIL**)
- Use safer functions that do **bounds checking**
- Use **automatic tools** to analyze code for potential unsafe functions.



So what if you could not use typeset languages? You could not use languages like Java or others that do this checking to prevent all memory overflow kind of attacks. What can we do in that case if you're stuck with let's say languages for whatever reason, we talked about a few, where this is not done automatically, what should we be doing? So when we are using these

so called unsafe languages that automatically don't prevent buffer overflows then it becomes the responsibility of the programmer. If you're writing code it is your responsibility that you deal with this problem of potential buffer overflows.

One way you can do that is by checking ALL input, okay? So secure coding mantra is that trust no input. All input is evil and you should be checking that it can conform to whatever your expectation was. Don't rely on the underlying system to do those because we're talking about using languages that do not

do automatically. So secure coding is an extremely as you write programs. You write them to implement certain functionality. You may even write them with some performance goals in mind, but at the same time you need to keep in mind that you have to write them so they cannot be exploited. You have to code securely or use secure coding practices so your program doesn't have the vulnerabilities like the ones we've been talking about. And one way you do that is by checking ALL input.

●Use safer functions that do **bounds checking** The other thing that you can do is, you can use functions that are safer that do bounds checking. So remember in the code we could have string compare or string copy. You could also have certain side string, though you shouldn't be dealing with a string that is longer than that. So their functions that would place a limit on the length of the string this function would manipulate. So you can use there number of safer ways to use certain functions that are unsafe and the patterns and ways of using those that are safer where the likelihood of a vulnerability is a lot lower. So these functions, all though the language doesn't do it, the functions that you do will do the checking or the bounds checking that we're talking about, and as a result using those leads to code that is safer more secure.

●Use **automatic tools** to analyze code for potential unsafe functions. And the last thing you can do after written some code that's not in a typeset language, is to use tools that exist that would actually analyze code that you have written and flag potential vulnerability. The way they able to do that is, they can look for certain code patterns. They can look for certain unsafe functions, and can warn you that the code fragment that you have where these things appear, an unsafe function appears, could be potentially vulnerable to some sort of exploit buffer overflow or something like that. So these tools, you just run your code through these tools and there's a lot of them that are out there. Some commercial, some you can get for free. And these tools they rely on sort of database of certain kind of patterns that are known to be unsafe. And if they run into your code that matches those kind of patterns, they can flag those. And they can do more sophisticated analysis to potentially discover the kind of vulnerability that we're talking about.

So there's no excuse for writing code that is not secure. You should be checking your input. You should be using safer functions that do that checking automatically. And then you should run through your coded through these tools that can flag potential problem spots. And you should go back and look at that and see how you can actually make those potential vulnerabilities go away. One problem with automatic tools is that they may have a lot of false positives or if they miss out and they have false negatives. So you can't have tools that will have zero false positives or false negatives, but the tools can certainly help us get rid of many possible vulnerabilities in the process producing code that is a lot more secure. So even when you are stuck with programming in a language that's not going to help you much with the kind of attacks, vulnerabilities that we've been talking about, there's certainly things you can do as a programmer who likes to do secure programming by checking input, using safer functions, and then running, checking your code through these tools that we're talking about. And we'll have a long list of these tools actually you can go and learn about.



Strongly vs. Weakly Typed Language Quiz

Strongly typed languages help reduce software vulnerabilities. Determine which of the following options apply to strongly typed languages and which are for weakly typed. (Use 's' or 'w').

- ☐ Any attempt to pass data of incompatible type is caught at compile time or generates an error at runtime.
- ☐ An array index operation $b[k]$ may be allowed even though k is outside the range of the array.
- ☐ It is impossible to do "pointer arithmetic" to access arbitrary area of memory.



Strongly vs. Weakly Typed Language Quiz

Strongly typed languages help reduce software vulnerabilities. Determine which of the following options apply to strongly typed languages and which are for weakly typed. (Use 's' or 'w').

- ☒ Any attempt to pass data of incompatible type is caught at compile time or generates an error at runtime.
- ☒ An array index operation $b[k]$ may be allowed even though k is outside the range of the array.
- ☒ It is impossible to do "pointer arithmetic" to access arbitrary area of memory.

Instructor Notes

https://www.owasp.org/index.php/Source_Code_Analysis_Tools

Defense Against Buffer Overflow Attacks



Analysis Tools...

- Can **flag** potentially unsafe functions/constructs
- Can **help mitigate security lapses**, but it is really hard to eliminate all buffer overflows.

Examples of analysis tools can be found at:

https://www.owasp.org/index.php/Source_Code_Analysis_Tools

So the tools that I was talking about you can find those owasp.org website. So remember we're talking about source code analysis tools here. So we are making assumption that the code that you write, you have the source code and you want to make sure that it doesn't have those vulnerabilities. So you going to run it, so you obviously have the source code. You run it through these tools.

A lot of times if the code is not written by you, it's coming from somebody else. You may only have the binary. And then some of those tools that requires source code, reduce source code analysis obviously won't be helpful. So discovering those vulnerability and binary is said reverse engineering and all the other skills come in, but attackers for example will take binary code and try to discover vulnerabilities in it. And we do that, one of the lab courses we teach here. People with attackers mindset, given a piece of binary how do you find out if has let's say, some potential overflow kind of vulnerability or not.

But if your source code, you writing the code and you want to deliver secure coding and so on. If you work for a company part of their software development process may include reviews and passing those codes through these kind of tools and so on. It's a problem that we all recognize, we all recognize the importance of writing secure code. And these analysis tools can help you identify potential vulnerabilities and mistakes that you may have made. That could result into these kind of things that we've been talking about. So, spending a little bit of time with these is a good idea and that's going to help you write more secure code.

Thwarting Buffer Overflow Attacks

Stack Canaries:

- When a return address is stored in a stack frame, a random **canary value** is written just before it. Any attempt to rewrite the address using buffer overflow will result in the canary being rewritten and an overflow will be detected.



We said buffer overflows are bad, so, what can we do beyond just relying on the programmers or coders to avoid those kind of vulnerabilities?

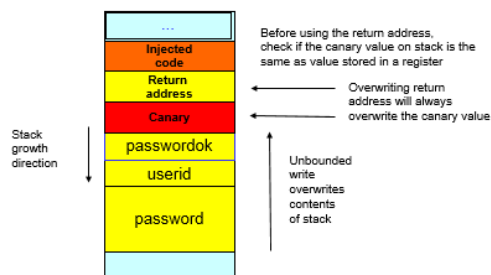
One way we can do that is, remember that one of the tricks that hacker uses is to override the return address field that we have on the stack, in a stack frame. Whatever the function you were executing when

the bad input has to received, the return address is modified. That should not be during the execution of that function. There is no reason for that return address to be modified because this return address is where we are supposed to go back. And where we go back should not change while we're executing this call function.

Okay, so how can you detect if the return address has been modified? And if it's been modified there is an overflow and if you can automatically detect that then you can stop things right there, program can terminate or something like that. So how can we detect any modifications?

Countermeasure – Stack Protection

Canary for tamper detection

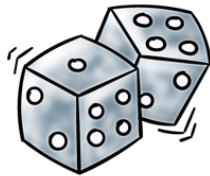


No code execution on stack

So one technique is what is called stack canaries. So these are sort of like coal mines. We have those birds that tell you when the harmful gas concentration is high or something like that. What we can do is just before the return address in a frame we can store when the function call is made. At that time we can store a canary value in a location that is just before the return address. If you're going to overflow into the return address, you're going to come wire this canary value. So if

the return address gets overwritten, chances are that the canary value is going to get overwritten also. So all you had to do is, before you return from the function, you check if the canary value has changed. You knew what the canary value is when the function call was made because that you had placed it in this location just before the return address. So you're going to what that location contains now with the value that you knew. If they don't match then there is an overflow and potentially the return address has been modified. And if it's been modified, then you shouldn't go on. We have detected a buffer overflow and that potentially could change program control flow and lead to attacker code and so on. So that modification should basically indicate to us that there is a problem here. So a canary value basically just says, if it had changed this is done automatically isn't it? The programmer has to do nothing. We compile it the way sets up the stack and sets up the data structure within the stack frame. That's where we add in this and we checking it automatically. So the programmer has to do nothing to get this extra security that comes from this defense that we just talked about that detects buffer overflows. So stack canary is one way that you can do that.

Thwarting Buffer Overflow Attacks



- **Address Space Layout Randomization (ASLR)** randomizes stack, heap, libc, etc. This makes it harder for the attacker to find important locations (e.g., libc function address).
- Use a **non-executable stack** coupled with ASLR. This solution uses OS/hardware support.

The couple of other techniques, in fact the next one we're going to talk about is hardly used. A number of operating systems now and so one thing that's going to be different about the next two techniques we'll talk about is that they going to use support from the operating system/hardware that we have to deal with buffer overflow or to thwart buffer overflow attacks.

- **Address Space Layout Randomization (ASLR)** randomizes stack, heap, libc, etc. This makes it harder for the attacker to find important locations (e.g., libc function address).

So the first one, as which I said many operating systems use, is what is called Address Space Layout Randomization, ASLR. So remember, in an address space, some place stack is space where stack is allocated. So, stack starts at certain place. Somewhere the library code goes, another place

where the heap goes, and so on. What if we randomize the places where these areas of memory start?

Remember that hacker had to actually guess where key information is. So if I wanted to do return to libc attack, I had to know where the library is, where a certain function may be, so I know the address of that function. And I can use that as I overflow the return address field in the stack frame. So, I need to know these address. And what ASLR does, it makes hard, because of the randomization that we do, it makes hard for the attacker to find these important locations. Where the address that the attacker is interested in is going to over the information, address of the information that that hacker is interested in is not known.

So as I said. A number of operating systems these days actually do that to protect the programs and processes from these buffer overflow kind of attacks. So this just makes it harder for the hacker to successfully craft an attack where can return, or manage, or direct control of the program to the certain known place that could be used to let the attacker execute arbitrary code.

- Use a **non-executable stack** coupled with ASLR. This solution uses OS/hardware support.
- There is one other defense against stack buffer overflow. When the shell code itself is stored on the stack. So remember if I don't do return to libc then my return address has to point me to some place where the shell code lives, or is stored, and you could write that shell code into some area of the stack itself. And then you can execute off the stack.

There's no legitimate reason for us to execute instructions that are stored on the stack. So one thing we can do is we can make the stack non executable. Basically says the system is not going to allow you to fetch instructions from the stack area or the stack segment. If you do that then you can't execute on the stack. So your shell code cannot be on the stack. You have to find some other place to put your shell code and maybe that's why you have to try tricks, like return to libc and things like that. And use a library function to transfer control to the code that attacker ones executed.

So here I guess we talked about sort of security is increasing assurance. Making the attacker's job more difficult. And making the legitimate, in this case here we're talking programmers, their task somewhat

easier. So ASLR obviously is making the attacker's job more difficult because key addresses that they need to know, they don't know exactly where they are in the address space. And they can't put code on the stack and hope to get it executed because we have the hardware operating system says not supposed to fetch instructions from the stack area of the address space. So these are techniques for actually dealing with buffer overflows. As we said, one makes it harder for the attacker, by making it difficult for him to find the right addresses, or be able to use the stack flow shell code.



Buffer Overflow Attacks Quiz

- Do **stack canaries** prevent return-to-libc buffer overflow attacks? ☐ Yes ☐ No
- Does **ASLR** protect against read-only buffer overflow attacks? ☐ Yes ☐ No
- Can the **OpenSSL heartbleed vulnerability** be avoided with non-executable stack? ☐ Yes ☐ No

Now that we have talked about number of defenses against buffer overflow attacks, let's look at some questions that help us understand what these defenses can do for us and what they cannot.

So the defenses that we talked about included stack canaries, address based loud randomization. So, the first two questions about that, and the third question is whether a certain buffer overflow vulnerability, read only kind in this case, would be

avoided, or would go away, when we can't execute code off the stack. So look at the defenses and what they can do for the kind of exploits we talked about before, and once you have a chance to do that we'll come back and talk with the answers.



Buffer Overflow Attacks Quiz

- Do **stack canaries** prevent return-to-libc buffer overflow attacks? ☒ Yes ☐ No
- Does **ASLR** protect against read-only buffer overflow attacks? ☐ Yes ☒ No
- Can the **OpenSSL heartbleed vulnerability** be avoided with non-executable stack? ☐ Yes ☒ No

So, the first question says, does the use of stack canary prevent return-to-libc buffer overflow attacks?

Remember return-to-libc buffer overflow relies on the ability to rewrite the return address in the stack frame. We rewrite it with the address of a library function that we returning to a library function. But we have to write or modify that return address.

And we said the use of a stack canary

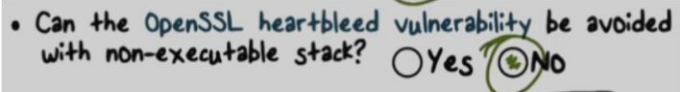
is to detect if the return address on the stack has been modified. Because when that gets modified, the canary value is going to be modified as well. So the answer here should be the yes, because a stack canary does allow you to detect change or modification to the return address, so that's why the answer is yes.

- Does **ASLR** protect against read-only buffer overflow attacks? ☐ Yes ☒ No

Second question, does ASLR protect against read-only buffer overflow attacks? Well, ASLR makes it hard to guess where certain addresses

of certain variables are, so for example a certain function in a library because library starts at some random address. When we talk about read-only buffer overflow attacks, we already have an address. We start reading from that address and then we keep going. We overflow into whatever is after it. So,

we're not trying to guess an address. We have an address and reading starting that and reading more than they should. But there isn't a problem that we have here of guessing an address where we should start. So because of that, ASLR does not protect us against read-only buffer overflows and that's why the answer is no.



• Can the OpenSSL heartbleed vulnerability be avoided with non-executable stack? ☐ Yes ☒ NO

The last one is actually read-only buffer overflow but that's the open SSL Heartbleed vulnerability. Can that be avoided with a

non-executable stack? Non-executable stack remember, is the code cannot be placed on the stack because it can't be executed, it can't fit instructions from the stack area. The answer to this particular question is no, because the OpenSSL Heartbleed vulnerability read more data. It didn't actually execute code. The overflow was reading certain data that was sensitive that it was not suppose to the function was not supposed to access it. So, there's no execution. We're not executing code to get to the sensitive data, and because of that it's had nothing to do with non-executable stack. So the vulnerability that over-reads or reads beyond the one that it's supposed to, we can't prevent that or make that go away just because we have non-executable stack.

So here we talked about a number of defenses and where they work and when they may not work. And going back, write a secure code, use a typeset language or use a program in a language that makes buffer overflows go away. If you can do that, you have to use a language where that's a possibility then check input. Use tools to look at potential places where there may be a vulnerability. And then hopefully, the system that you're working on has defenses of type such as ASLR for which you don't need to do anything as a programmer. Or things like stack protection where you can't execute off the stack.

Software Security

Lesson Summary

- Understand how software **bug/ vulnerabilities can be exploited**
- **Several defenses** possible against attacks
- **Buffer overflows** remain a problem
- **Web security:** Important for web
- **Secure coding -- check all input!**

We saw many of the common software vulnerabilities or bugs and how they can be exploited. We also learned about a number of defenses, including what we can do and also what the operating system can do for us. At this point, hopefully, you are well aware of some of the secure programming practices that we should all be using.