Introduction to Buffer Overflows

Author: Jedidiah R. Crandall, crandaj@erau.edu

This Document was Funded by the National Science Foundation
Federal Cyber Service Scholarship For Service Program:
Grant No. 0113627

Distributed July 2002

Embry-Riddle Aeronautical University • Prescott, Arizona • USA

Basics of Buffer Overflows

What's a buffer? An overflow? How do attacks occur?

ANALOGY: computer~~mailroom

Scenario Characters:

- Norman, the mailroom worker
- Alice, who programs the mailroom computer
- Patty, an innocent-appearing user who knows how to use a buffer overflow to hijack the computer

What is a *buffer*?

An example of a buffer:

- A place to fill in your last name on a form where each character has one box. **SMITH** fills five boxes, **GALLERDO** fills eight.

Techie Nomenclature: a “buffer is used loosely to refer to any area of memory where more than on piece of data are stored.

What is a buffer *overflow*?

Imagine the “last name” field has 10 boxes. Your last name is **HEISSENBUTTEL** (13 characters). Refusing to truncate your proud name, you write all 13 characters. The three extra characters have to go somewhere!

A computer buffer overflow – attacker treachery ahead

A computer allocates a buffer of memory to store ten integers. An attacker gives the computer 11 integers as input. Whatever was right after the buffer in memory gets overwritten with the 11th integer.

Techie nomenclature: actually an integer is stored in 4 bytes, but that's not significant to the story.

Why is this a security problem?

Overflowed data could be anything.

- An integer where “0” means that you <u>can’t</u> access a particular file and and “1” means you <u>can</u>. A hacker would overwrite the “0” with a “1” and access the file.
- Characters like **ROOT (A HIGHLY PRIVILEGED USERS)**
- A *pointer* that tells the program what instructions to execute next.
- Even a minor change could cause the program to crash which can be a security problem (*denial-of-service* attacks and *core dump* exploits are very serious).

How could a computer (or programmer) be so dumb?

- The computer (or programmer) should check the size of the buffer first before trying to put all of the data into it.
- Popular languages like C/C++ don't automatically check the bounds of the buffer.
- Programmers who use C/C++ are responsible for performing this check. Often they don't.
- Programming shops often don't use checklists to spot this type of error and often testers don't think of trying to make buffer overflows show up

Answer: Modern software practice is sloppy, and buffer overflows get through (see the life cycle).

- The problem can be a lot more complex when you start talking about supporting international character sets and copying/formatting/processing buffers that store different kinds of things.

Want a more detailed but non-techie explanation?

ANALOGY: computer~~mailroom

Scenario Characters:

- Norman, the mailroom worker
- Alice, who programs the mailroom computer
- Patty, an innocent-appearing user who knows how to use a buffer overflow to hijack the computer

ANALOGY: Computer~~Mailroom

Imagine a mailroom with 256 mailboxes, numbered 0 through 255.

Each mailbox can contain only a single piece of paper with a number from 0 through 255 written on it.

Why 256? Techie nomenclature:
One *byte* = 8 *bits*. A bit can be either “0” or “1”.
2^8 = 256 possible combinations of 8 bits. So every number can be represented.

New data overwrites previous data

IMPORTANT:

If Norman, the mailroom worker, puts a piece of paper with a number on it in a mailbox, and that mailbox already has a piece of paper, he takes the old piece of paper out and throws it away. Then he puts the new piece of paper in its place. The old data is overwritten with the new.

Big numbers take multiple mailboxes

- If someone wants to store a number bigger than 255, they need multiple mailboxes.
- Four mailboxes means 8 bits x 4 mailboxes = 32 bits. 2^{32} = 4,294,967,296 so you can store numbers in the billions with four mailboxes.

Characters are represented as numbers, too

The easiest way to store something like a name is to assign each number from 0 to 255 with a character.

For example: “A” = 65, “D” = 68, “d” = 100.

One such assignment is the (Technie nomenclature) *ASCII* character set which is a national standard.

Programs end up as numbers in mailboxes, also!

Programs are stored the same way, as a buffer of numbers. A program that reads someone's last name and repeats it to them is *compiled* into a series of *instructions* that Norman understands.

Each instruction has a unique number. For example,

- 23 = "Copy the number from the next mailbox and hold that piece of paper in your left hand."
- 188 = "Put the piece of paper in your left hand into the mailbox with the box number written on the piece of paper in your right hand."

.

Special numbers used as addresses are also stored in mailboxes!

If a number, such as the one in his right hand in this example, is used to reference a mailbox by its *address* then that number is referred to as a *pointer.*

You guessed it, these pointers are also stored in mailboxes.

Programs use addresses for commonly used orders and to change order of actions

Norman can only read an instruction from one mailbox, *execute* it, and then move onto the next mailbox. But sometimes the program wants him to *jump* to a new address, in which case the program gives him a pointer to the next instruction he should execute (instead of just the one in the next mailbox).

- The pointer is just a number which is the address of another mailbox.
- The *program counter* is the number that the mailroom worker has to remember in order to remember what mailbox's instruction he is executing at any moment.

The mailroom is a computer.

Computers need programmers to give them orders.

Norman, the mailroom worker, is the computer and the wall of mailboxes is the memory.

Alice the programmer starts by filling some mailboxes with instructions. Then she tells Norman which mailbox to start with and leaves him alone until he comes back to her and says he's finished.

Really! The mailroom is just like a modern computer on another scale!

In fact, the only real difference between the mailroom and a real computer is that a real computer might have dozens of hands (called registers) and the memory might have millions of mailboxes (called words).

Consequently, a pointer to a word in a computer's memory might need to be in the billions and therefore you would need 4 bytes of memory to store it.

Programmers use subroutines for common functions used many places

Alice breaks her program into *subroutines*.

Subroutine R might invoke (or call) subroutine S which invokes (calls) subroutine T. When T is finished Norman needs to resume with subroutine S, and when S is finished he needs to resume where he left off in subroutine R.

Executing subroutines requires stacks to keep track of returns

A stack is like a stack of trays in a cafeteria. You can only put one on the top or take one off the top. You can't touch any in the middle or on the bottom.

- If Norman is executing subroutine R and the instruction at mailbox number 37 tells him to call subroutine S at mailbox 82, he writes 37 on a tray and puts (or *pushes*) it on the stack. This tray is called a *return pointer*. Then he can jump to mailbox 82.
- Then the instruction in mailbox 83 tells him to call subroutine T at mailbox 112 so he writes 83 on a tray and puts (pushes) it on a stack.
- When subroutine T is finished he grabs the tray off top of the stack (or *pops* it), sees the 83 written on it, and knows where to resume (84).
- When S is finished he pops another tray, sees the 37, and can resume where he left off with R (at 38).

Stacks are essential (TBD omit)

The reason *C* compilers use stacks is that subroutines can call subroutines that call other subroutines.

- Subroutines can call each other, and subroutines can even call themselves, and you can always keep track of where you were in each subroutine by pushing and popping on and off a stack.

Problem: Pointers to stacks need mailboxes

Norman doesn’t have any trays. He has to use pieces of paper and mailboxes and has to remember where the top of the stack is at any moment.

If he remembers that the top of the stack is 212, then he can pop a byte by taking the piece of paper out of mailbox 212, but then he has to remember that the top of the stack is now 211.

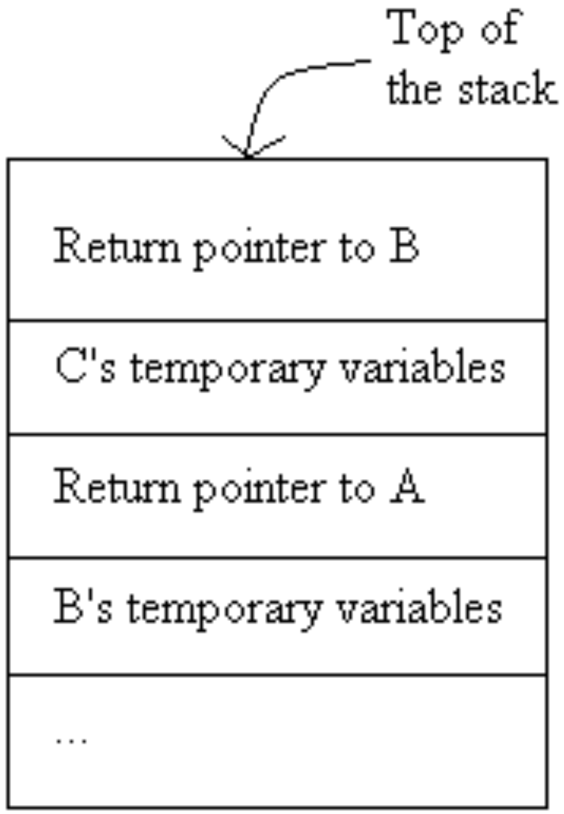
Problem: Subroutine data also needs to go on and off the stack, too

Alice wants other things to go on the stack, too, like temporary buffers for storing data that only subroutine S needs.

That way, when subroutine S returns execution to R the return pointer is popped off the stack, but also on the stack is all of S's temporary memory which needs to be taken off the stack and destroyed.

What stacks look like

The stack after A calls B and B calls C:



All the pieces are in place – one minor error and we'll see how to hijack the mailroom.

Part of Alice's instructions to Norman was for him to ask Patty the user for her last name (as a series of numbers on pieces of paper) and store it in a temporary buffer on the stack. These instructions are encapsulated in a subroutine called GetLastName().

How to hijack the mailroom: the stack arrangement is the culprit

When the GetLastName() subroutine is called, a buffer is put on the stack for its use. The buffer is an array of 10 mailboxes to store 10 characters for Patty’s last name.

An 11th character is needed for the return address, to tell Norman where to resume execution after he’s finished with GetLastName().

Patty wants into that computer!

Pretend that the part of the stack with these two things on it starts at mailbox 68. Therefore, mailboxes 68 through 77 store the 10-character last name, and mailbox 78 has the return address.

Patty the user is actually Patty the attacker, and goes by the last name **SOLZHENGRAD**.

Alice goofed, a common mistake, and Patty knows how to take advantage.

Alice didn’t write the program so that it checked to make sure the last name was 10 characters or less before the mailroom worker tried to store it.

Norman is simple-minded; he only does what he is explicitly told to do.

Patty knows where Norman will store her name and that part of her name is a program

Patty the hacker gives Norman 11 pieces of paper, each with a number representing an ASCII character:

- **SOLZHENGRAD** is stored as 83-79-76-90-72-69-78-71-82-65-68.

Norman starts at mailbox 68 and replaces whatever was there before with the new piece of paper. 83➔mailbox #68, 79➔mailbox #69, 76➔mailbox #70, 90➔mailbox #71, etc.

Patty knows that a buffer overflow will ovewrite the return point to go to her program

68, the ASCII representation for a “D”, is put in mailbox #78. But mailbox #78 was being used to store the return address!

Now when Norman is finished with the GetLastName() subroutine, he will read a 68 as the return address instead of the real return address.

He’ll then begin executing instructions at mailbox #68, which is Patty’s instructions to steal Norman and make him do what Patty wants instead of what Alice wants (Patty’s hijacking subroutine, if treated as ASCII, just so happens to spell out **SOLZHENGRA**).

Glossary

Buffer – A buffer is an area in a computers memory to store data. If it is a single piece of data, such as a number or a single character, then its storage space is usually not referred to as a buffer. The real definition of a buffer is somewhere where data is stored temporarily, but the term buffer is often used more loosely.

Pointer – Pointers point to something in the computers memory. Everything stored in a computer is stored as a number, including pointers. A pointer is a number that is the references another place in memory by its address.

Denial-of-service – Sometimes if a program is needed by multiple users (on a network, for example) and an attacker crashes it, no one else can access it.

Core dump – A core dump occurs sometimes when a program crashes. Basically, everything that was in that program's memory is written out to an unprotected file, and sometimes this data is security-sensitive.

Address – An address in a computer's memory is the same as the address in a mailbox. If your box number in the mailroom is 232, then 232 is called your address. The same is true for data and programs stored in a computer's memory.

Bit – A computer stores numbers using bits. A bit can be only one of two things: a 1 or a 0.

Byte – A byte is an 8 bit number, such as 10011110. A byte can store a number from 0 through 255, or 00000000 through 11111111 in binary.

Glossary

Compiled – A compiler takes the source code a programmer has written and turns it into code text that a computer can easily execute. Code text is a sequence of instructions for the computer stored in memory.

Instruction – An instruction tells the computer what to do, but it is very low-level. Unlike source code, it works at the microprocessor level telling the computer what elementary steps to take to execute the program.

Source code – this is the code that the programmer writes in a high-level language like Java, BASIC, or, in this case, C. A high-level language is one that humans can easily read and understand.

Execute – To execute the program means to “run” it. The two terms can be used interchangeably. To execute an instruction means to do it. It is the process of seeing what the instruction is, and then doing it.

Jump – A jump occurs when the computer is told to start executing instructions somewhere else besides the next instruction after the current one being executed.

Program counter – The instructions are stored as a sequence of numbers in the computer just like anything else. The computer usually executes one instruction after another unless it is told to *jump* somewhere else. The program counter keeps track of what instruction in memory is being executed at any given time.

Subroutine – Good programmers break they’re program up into smaller steps called subroutines. A program to make coffee could be broken up into preparing the filter, putting water in the coffee maker, and turning the coffee maker on. Each of these subroutines can be broken down into more subroutines. Putting water in the coffee maker could be broken down into get a cup, turn on the sink, put the cup under the sink, turn off the sink when the cup is full…

Glossary

Push – Pushing is the operation of putting something new on the top of a stack. Computers use pushing to store the return pointer when one subroutine calls another subroutine.

Pop – Popping is the operation of taking something off the top of a stack. Computers use pop operations to retrieve the stored return pointer and use it as a reference to get back to the original subroutine that made the call to the subroutine that just finished.

Return pointer – A return pointer is a special kind of pointer that computers use on a stack to remember what instruction they were about to execute in one subroutine when they had to go start executing another subroutine.

C – C is the most commonly used high-level computer programming language, and the one most responsible for the buffer overflow problem. C source code is compiled into a low-level language that a computer can understand, such as a bunch of pieces of paper with numbers written on them in mailboxes.

About this Project

This presentation is part of a larger package of materials on buffer overflow vulnerabilities, defenses, and software practices.
For more information, go to: http://nsfsecurity.pr.erau.edu

Also available are:

- Demonstrations of how buffer overflows occur (Java applets)
- PowerPoint lecture-style presentations on an introduction to buffer overflows, preventing buffer overflows (for C programmers), and a case study of Code Red
- Checklists and Points to Remember for C Programmers
- An interactive module and quiz set with alternative paths for journalists/analysts and IT managers as well as programmers and testers
- A scavenger hunt on implications of the buffer overflow vulnerability

Please complete a feedback form at http://nsfsecurity.pr.erau.edu/feedback.html to tell us how you used this material and to offer suggestions for improvements.