Buffer Overflow Defenses

Some examples, pros, and cons of various defenses against buffer overflows.

Caveats:

1. Not intended to be a complete list of products that defend against buffer overflows.
2. There is no silver bullet that will stamp out buffer overflows, but some of these tools may help.

Kinds of Defenses

- Better software engineering practices
- Find-and-patch methods
- Language tools
- Analysis tools
- Compiler tools
- Operating system tools

Better Software Engineering Practices

- Testing
  - Execution of the software with selected data.
- Code Inspection
  - Inspection of the code by humans with a checklist to make sure the code meets certain criteria.
- Documentation of vendor code
  - Documentation of vendor code components that others may reuse in their own projects.

Better Software Engineering Practices - Testing

Pros:

- Good testing practices should catch most buffer overflows

Cons:

- Time is money, sometimes it is a more economically sound solution to allow buffer overflows than to find them
- When using vendor software, you cannot *white-box* test software that you do not have the source code or the documentation for
- Data corruption is harder to detect than abnormal program behavior without dynamic analysis tools

Better Software Engineering Practices- Code Inspection

Pros:

- Code inspection may catch many buffer overflows that testing won't

Cons:

- Time is money
- When using vendor software, you cannot do a code inspection if you do not have the source code

Better Software Engineering Practices - Documentation

Pros:

- Good documentation of reusable software components will allow people who use your code in their own projects to test and inspect it

Cons:

- Time is money, and the cost of documenting the code gets passed on to the customers
- Often software companies do not want to release the source code for libraries that they sell

Find-and-patch Methods

- Software patches
  - released by vendors when a security problem in their software is found, to fix the vulnerability.
- Programs that block known attacks
  - Programs that keep a list of known attacks and watch for those attacks on your system.

Find-and-patch Methods – Software patches

Example: The vendor, the customer, or a group concerned about software security finds a buffer overflow and a patch is written and released

Pros:

- Very effective at preventing known buffer overflow attacks for specific vulnerabilities

Cons:

- No protection against unknown attacks or known attacks for which a patch has not been released
- Not all patches fix the buffer overflow, some are specific to one attack but leave the buffer overflow itself in place
- The customer must regularly check for patches for their system (at the vendor's website or www.cert.org) and install them.

Find-and-patch Methods– Programs that block known attacks

Example: An anti-virus program that checks files and other inputs to the system for signatures of known attacks

Pros:

- Very effective against specific attacks that are known

Cons:

- Not effective against unknown attacks or attacks for which the anti-virus program does not yet have the signature
- The program must keep a current list of signatures for known attacks and must be updated regularly

Language tools

- Languages less susceptible to buffer overflows
  - Languages other than C/C++ that are less susceptible to buffer overflows when used properly.
- Languages based on C
  - Languages like Cyclone that were designed with preventing buffer overflows in mind.
- "Safe" buffers
  - Buffers that automatically truncate inputs, generate exceptions, grow bigger.
- Safer library functions
  - Library functions that are less susceptible to buffer overflows than the standard C library.

Language tools – Languages less susceptible to buffer overflows

Examples: Ada, Java, Perl, Python, etc.

Pros:

- Automatic bounds checking makes them less susceptible to the buffer overflow problem
- Exception handling can greatly ameliorate the problem

Cons:

- Using different languages can increase development cost
- None of these languages give the programmer access to the machine at a low level
- None of these languages give you the performance of C/C++, most require distributable run-time environments
- C/C++ are popular languages that many programmers are familiar with
- What happens when a string that is too long is entered or an array is referenced out of bounds, is an exception generated, does the buffer grow, does the program just halt, is the user asked to provide different input?
- Programmer still must be aware of buffer overflows to provide exception handlers to do what they want (Exception handling comes with its own set of problems)

Language tools – Languages based on C

Example: Cyclone is a different dialect of C that handles pointers in a much safer manner

Pros:

- The transition from C to Cyclone is an easy one because Cyclone is nearly identical to C

Cons:

- Existing C source code must be recompiled and probably modified
- Code ported to Cyclone must be debugged, and gdb (a commonly used UNIX-based debugger) does not work well with Cyclone
- Using pointers in Cyclone is considerably more complicated than using pointers in C ('*' is replaced with '*', '@', and '?')
- Cyclone does not provide object-oriented features

Language tools – "Safe" buffers

Example: C++ class objects that do bounds checking like CString, or "limitless" strings like libmib

Pros:

- Much safer than standard string handling in C
- Exceptions can be handled instead of a program halt

Cons:

- Require the use of different library functions, meaning that existing code has to be modified or interfaced with in a low-level way
- A "limitless" string has to continually be reallocated meaning a bigger heap and a performance cost
- What if you do not want the buffer to grow and accept a bigger input?

Language tools – Safer library functions

Example: Use of a different library than the standard C libraries

Pros:

- Eliminates problems with unsafe library function calls in C/C++

Cons:

- Existing code has to be modified
- Programmers have to become familiar with a different set of libraries
- Often string and memory handling libraries are replaced, but not standard library functions specific to an operating system, like file handling and environment variable functions which can also lead to buffer overflows
- Not all buffer overflows are caused by library functions
- What happens when a buffer's limit is reached? Does the program halt? Is the string truncated? Is an exception generated?

Analysis tools

- Static analysis – Tools that find possible defects in the source code.
- Dynamic analysis – Tools that find possible defects by analyzing things like memory usage during execution of the program.

Analysis tools - Static

Examples: Software that searches source code for unsafe library function calls like ITS4

Pros:

- Can be a very effective tool during code inspection by finding unsafe library function calls and making recommendations

Cons:

- Only effective against buffer overflows caused by unsafe standard C library function calls
- Produces many false positives, only a fraction of the library function calls that are reported are actually unsafe

Analysis tools - Dynamic

Examples: Tools that analyze memory use of a program during testing, like Purify

Pros:

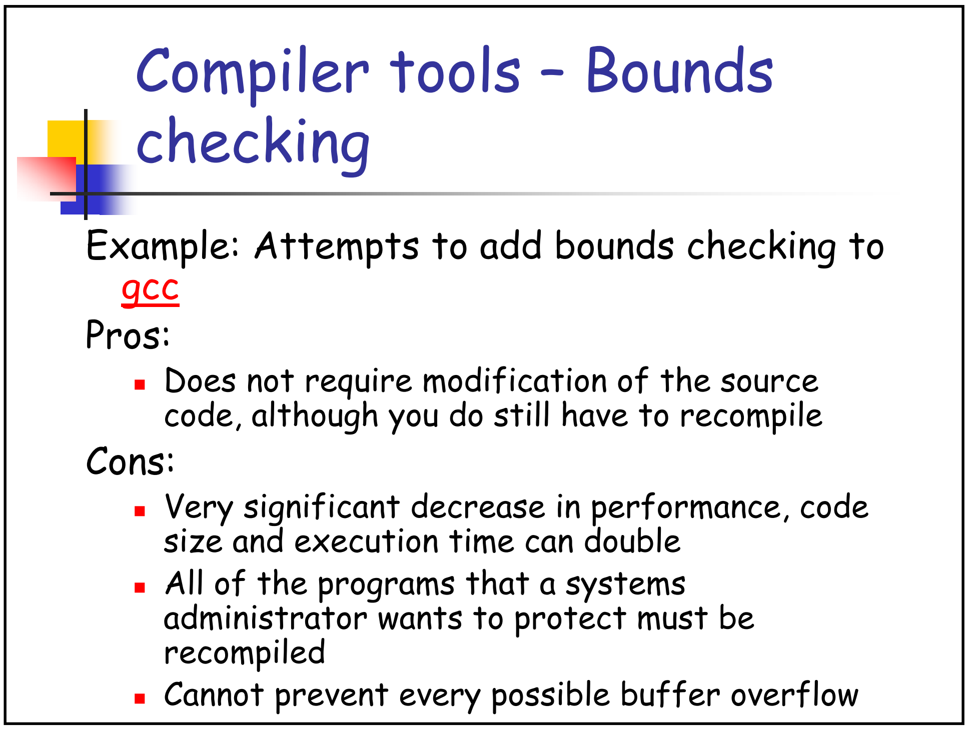
- Can detect buffer overflows that occur during testing
- Sometimes testing will not catch buffer overflows where data is corrupted but program behavior is not affected, dynamic analysis will

Cons:

- Buffer overflows that lead to erratic program behavior can usually be found during testing without dynamic analysis tools

Compiler tools

- Add bounds checking to all buffers
- Protect the return pointer on the stack



Compiler tools – Bounds checking

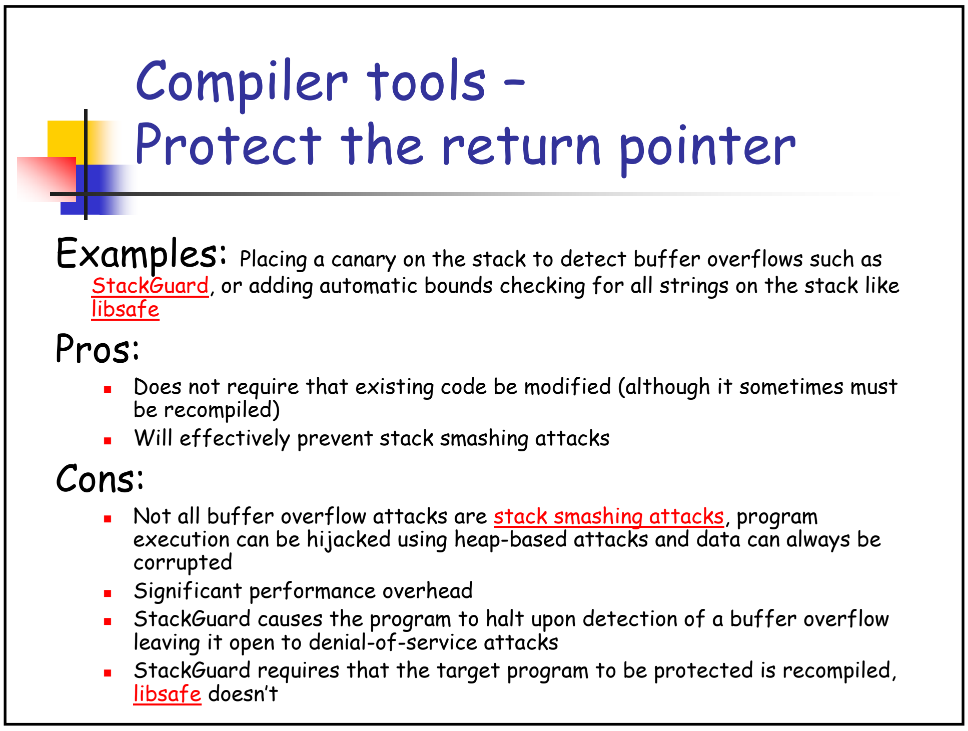
Example: Attempts to add bounds checking to gcc

Pros:

- Does not require modification of the source code, although you do still have to recompile

Cons:

- Very significant decrease in performance, code size and execution time can double
- All of the programs that a systems administrator wants to protect must be recompiled
- Cannot prevent every possible buffer overflow



Compiler tools – Protect the return pointer

Examples: Placing a canary on the stack to detect buffer overflows such as StackGuard, or adding automatic bounds checking for all strings on the stack like libsafe

Pros:

- Does not require that existing code be modified (although it sometimes must be recompiled)
- Will effectively prevent stack smashing attacks

Cons:

- Not all buffer overflow attacks are stack smashing attacks, program execution can be hijacked using heap-based attacks and data can always be corrupted
- Significant performance overhead
- StackGuard causes the program to halt upon detection of a buffer overflow leaving it open to denial-of-service attacks
- StackGuard requires that the target program to be protected is recompiled, libsafe doesn't

Operating system tools

- Disable execution of code outside the code space
  - It is possible on some architectures to distinguish between code and data, and not allow data to be executed as code.
- Intrusion detection systems
  - These are programs that watch for abnormal behavior or behavior that is similar to attack behavior.
- Generation of an interrupt
  - With hardware support it is possible to set bounds on a buffer and generate an interrupt when an attempt is made to access or change memory outside those bounds.

OS tools – Disable code execution outside the code space

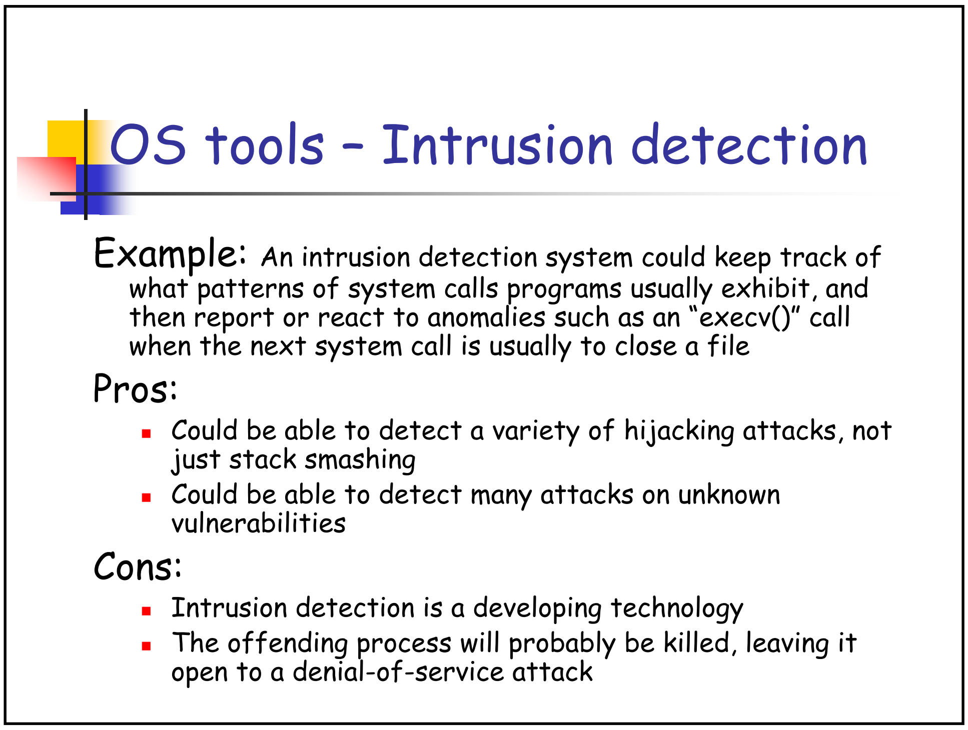
Example: A patch for Linux that disables execution of code on the stack as well as maps library function calls to addresses with a zero byte in them

Pros:

- Currently, the most common and most devastating buffer overflow exploit is stack smashing and this patch makes stack smashing much more difficult
- Does not require that existing software be modified or recompiled
- A zero byte in the address of a system call forces the attacker to have a null character in the attack string

Cons:

- Does not prevent all stack smashing attacks, often attack code can be placed in global variables or on the heap, or library code to spin a shell already exists in the code space (i.e., system() or execv())
- Crashing still leaves programs open to denial-of-service and core dump attacks
- A null character in just the right place in an attack string is not always impossible for an attacker to accomplish, and they can always jump to a small piece of code in variable space that contains a second jump to the desired location
- Some legitimate programs execute code on the stack, but very few, and there is a work-around for this



OS tools – Intrusion detection

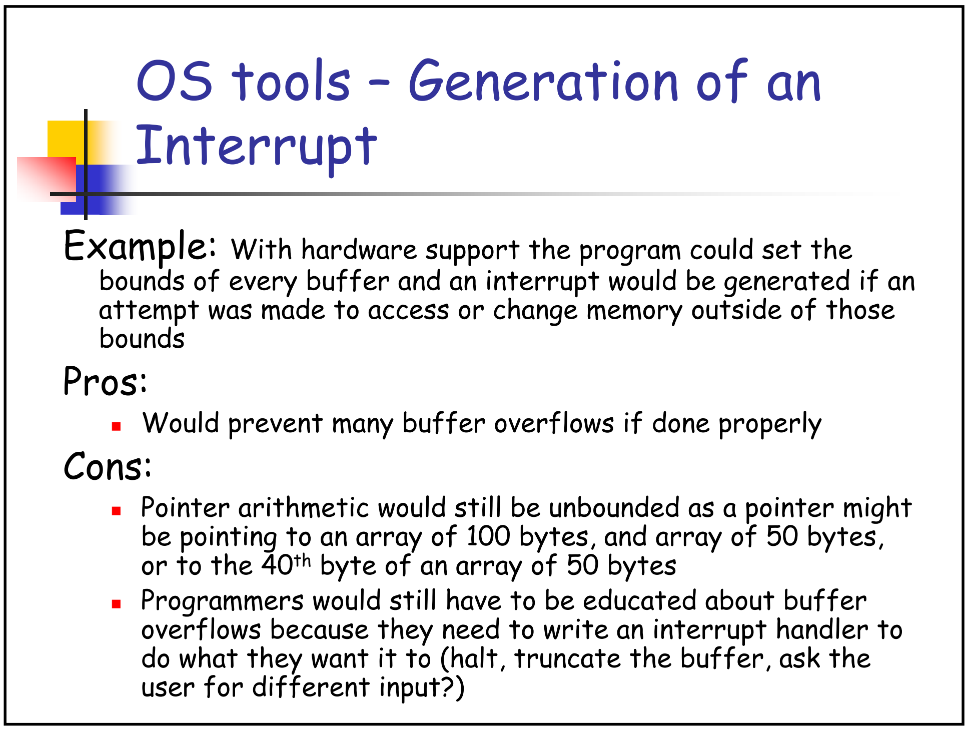
Example: An intrusion detection system could keep track of what patterns of system calls programs usually exhibit, and then report or react to anomalies such as an “execv()” call when the next system call is usually to close a file

Pros:

- Could be able to detect a variety of hijacking attacks, not just stack smashing
- Could be able to detect many attacks on unknown vulnerabilities

Cons:

- Intrusion detection is a developing technology
- The offending process will probably be killed, leaving it open to a denial-of-service attack



OS tools – Generation of an Interrupt

Example: With hardware support the program could set the bounds of every buffer and an interrupt would be generated if an attempt was made to access or change memory outside of those bounds

Pros:

- Would prevent many buffer overflows if done properly

Cons:

- Pointer arithmetic would still be unbounded as a pointer might be pointing to an array of 100 bytes, and array of 50 bytes, or to the 40th byte of an array of 50 bytes
- Programmers would still have to be educated about buffer overflows because they need to write an interrupt handler to do what they want it to (halt, truncate the buffer, ask the user for different input?)