Trusting Trust

Overview

Key idea:

- To believe you are 100% secure, you have to trust every piece of software and hardware that has ever been used to create your system

Main example: compilers with trojan horses

C compiler example

- Modify the source of the C compiler to contain two trojans:
  - the first will incorrectly compile the UNIX login command, inserting a backdoor
  - the second will incorrectly compile the C compiler, inserting these two trojans (similar to a self-reproducing program)

C compiler example

- Now use the official C compiler to compile the new source.
  - The new compiler will now insert a backdoor into UNIX, and it will also insert the above two trojans into the C compiler.
  - Making this the official C compiler will put backdoors in everyone’s UNIX.

Old quiz questions (#8, Q3 2010)

Indicate the truth or falsehood of each of the following with respect to Ken Thompson's paper "Reflections on Trusting Trust."

A. Thompson believes that self-reproducing programs shouldn't be trusted.

B. A Trojan horse like the one Thompson describes could not have been hidden in a compiler for a more modern language like Java.

C. The Trojan horse Thompson embedded in the login program could have been found by looking at the machine instructions being executed by the CPU.

D. A programmer can prevent the type of attack Thompson describes by writing all of his or her programs in assembly code.

Old quiz questions (#8, Q3 2010)

Indicate the truth or falsehood of each of the following with respect to Ken Thompson's paper "Reflections on Trusting Trust."

A. Thompson believes that self-reproducing programs shouldn't be trusted.

Answer: False. While his "trusting trust" attack hides in programs that produce programs, he doesn't say anything about this making them more or less trustworthy.

B. A Trojan horse like the one Thompson describes could not have been hidden in a compiler for a more modern language like Java.

Answer: False. There is nothing language specific about this attack.

C. The Trojan horse Thompson embedded in the login program could have been found by looking at the machine instructions being executed by the CPU.

Answer: True. It might be difficult, but the back door does appear in the binary executable code.

D. A programmer can prevent the type of attack Thompson describes by writing all of his or her programs in assembly code.

Answer: False. It would be just as easy to create a similar bug in the assembler used to translate the assembly code to machine code.

Old quiz questions (#5, Q3 2008)

You've written the source code of a compiler C for a new system. You compiler is written in the language Blub, and compiles the same language. Since you're a good MIT student, your compiler is completely bug-free.

Now you need to compile your source. You're given two executable Blub compilers, a and b, on the new system, but warned that one might be buggy or contain a Trojan horse. Except for this potential problem in a or b, the three compilers all implement Blub according to its specification, which is complete and exact.

You compile C using both a and b to produce executables ca and cb respectively. You then compile C using ca to produce cca, and compile C using cb to produce ccb. Finally, you compile C using cca to produce ccca, and compile C using ccb to produce cccb. You now compare the binaries produced at various stages of the compilation. Which of the following statements is true?

A. If ca and cb differ then one of a and b is necessarily buggy/Trojaned.

B. If cca and ccb differ then one of a and b is necessarily buggy/Trojaned.

C. If ccca and cccb differ then one of a and b is necessarily buggy/Trojaned.

D. If cca and ccca differ then a is necessarily buggy/Trojaned

E. If b contains a Trojan horse, then ccb and cccb will necessarily differ.

Old quiz questions (#5, Q3 2008)

A. If ca and cb differ then one of a and b is necessarily buggy/Trojaned.

FALSE. Even if a and b are bug-free, they will likely compile the same input code to different output machine instructions.

B. If cca and ccb differ then one of a and b is necessarily buggy/Trojaned.

TRUE. C should always produce the same output for a given input. That is, any two correct compilations of C's source should behave the same.

C. If ccca and cccb differ then one of a and b is necessarily buggy/Trojaned.

TRUE

D. If cca and ccca differ then a is necessarily buggy/Trojaned

TRUE

E. If b contains a Trojan horse, then ccb and cccb will necessarily differ.

FALSE. Perhaps the operation of the Trojan horse is not triggered in this particular case.

Beyond Stack Smashing

Overview

Types of exploits mentioned:

- Stack smashing
- Arc injection
- Pointer subterfuge
- Heap smashing

All of these use a buffer overrun

- Read/write past the end of a buffer

Stack smashing

- Overwrite a return address to point to code in a stack buffer (supplied by attacker)

- trampolining:
  - attacker doesn’t know buffer’s address. uses a “trampoline” -- instructions that transfer control using a pointer already in a register or on the stack

Arc injection

A.k.a. “return-into-libc”

- Transfer control to code that already exists in the program’s memory
  - e.g. overwrite a return address on the stack with a pointer to a specific part of the system() function in the C standard library (libc)
    - may need to prepare registers beforehand
- Useful when stack is nonexecutable

Pointer subterfuge

- Exploits modifying a pointer’s value
- Several varieties:
  - function-pointer clobbering
  - data-pointer modification
  - exception-handler hijacking
  - virtual pointer smashing

Pointer subterfuge

- Function-pointer clobbering
  - modifying a function pointer to point to attacker code
  - avoids mechanisms that protect return addresses
- Data-pointer modification
  - modifying data pointers to help in an attack
    - controlling the value of variables in a statement

      such as “*ptr = val” allows for an arbitrary memory write

Pointer subterfuge

- Exception-handler hijacking
  - making exception handlers invoke attacker code
    - examples (for Windows Structured Exception Handling):
      - There is a linked list of exception handlers, and the entries are stored on the stack
      - Can clobber a function pointer in this list
      - Can change the pointer to the exception handler list point to a fake exception handler list made by the attacker

Pointer subterfuge

- Virtual pointer (VPTR) smashing
  - C++ implements virtual functions by associating a virtual function table (VTBL) with each class -- this table is an array of function pointers. Objects have a virtual pointer (VPTR), which points to a VTBL.
  - Replacing a VPTR with a pointer to the attacker's fake VTBL transfers control when a virtual function is invoked.

Heap smashing

- Buffer overruns in the heap (dynamically allocated memory), instead of the stack
  - key idea: violate invariants maintained by the allocator
  - example: the allocator might keep headers for each allocated block. overwriting pointers in a header could result in an arbitrary memory write when that block is freed.

Old quiz questions (#5, Q3 2012)

Recall the buffer overflows paper from recitation, and consider the following two methods for countering buffer overrun:

• Non-executable stack: This method disallows execution from the memory region of the stack.

• Stack Canary: This method inserts a random value, which is hard for an attacker to guess or obtain, in the stack just before each function return pointer. Thus, if a buffer overflow occurs in the function, it overwrites the canary value before it overwrites the function return pointer. The canary value is checked before returning from the function to make sure it has not changed. If it did, the return pointer is not used.

Which of the following are true?

A. Stack canaries will protect against arc injection attacks that chain multiple function calls, as described in the paper.

B. A non-executable stack will protect against function-pointer clobbering.

C. Stack canaries will protect against function pointer clobbering.

D. Stack canaries will protect against data pointer modification.

Old quiz questions (#5, Q3 2012)

A. Stack canaries will protect against arc injection attacks that chain multiple function calls, as described in the paper.

Answer: True. Multiple function calls require return instructions which will catch possible stack buffer overflows.

B. A non-executable stack will protect against function-pointer clobbering.

Answer: False. An adversary can still clobber a function pointer that's not on the stack (e.g., a global struct).

C. Stack canaries will protect against function pointer clobbering.

Answer: False. Same as above.

D. Stack canaries will protect against data pointer modification.

Answer: False. Same as above.

Old quiz questions (#2, Q3 2011)

Based on the paper "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns", which of the following statements are correct?

A. The basic form of attack described in the paper (stack smashing) allows an attacker to execute arbitrary code by overwriting far enough beyond the end of an application buffer to also overwrite the stack pointer.

B. Even if a system guards against buffer overruns by making the stack non-executable, and uses stack cookies to guard against arc injection / return-into-libc attacks, the system could still be vulnerable to attacks which modify function pointers.

C. A heap buffer overrun writes beyond the end of a dynamically-allocated object (e.g., created with new in C++ or malloc() in C), instead of a procedure's local variables.

Old quiz questions (#2, Q3 2011)

Based on the paper "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns", which of the following statements are correct?

A. The basic form of attack described in the paper (stack smashing) allows an attacker to execute arbitrary code by overwriting far enough beyond the end of an application buffer to also overwrite the stack pointer.

Answer: False: it overwrites the return address, not the stack pointer.

B. Even if a system guards against buffer overruns by making the stack non-executable, and uses stack cookies to guard against arc injection / return-into-libc attacks, the system could still be vulnerable to attacks which modify function pointers.

Answer: True: function pointers could exist in local variables without needing to overwrite a cookie value.

C. A heap buffer overrun writes beyond the end of a dynamically-allocated object (e.g., created with new in C++ or malloc() in C), instead of a procedure's local variables.

Answer: True