Software Security

Buffer Overflows

more countermeasures

Erik Poll

Digital Security

Radboud University Nijmegen

Recap last week

- Recurring security problems in C(++) code
  - memory corruption, due to buffer overflows & bugs with pointers esp. using dynamically allocated memory (aka the heap)
  - integer overflows also as way to trigger buffer overflows
  - format string attacks for calls of `printf()` family
- *Spotting buffer overflows in C(++) code is hard!*
- Platform level defences:

  canaries, non-executable stacks, ASLR, CFI, bound checking with fat pointers, pointer encryption, ...

  against ever more advanced attack techniques:

  incl. return-to-libc & ROP

Common anti-patterns

Buffer overflows involve three more general anti-patterns:

1. **lack of input validation**

2. **mixing data & code**
   namely data and return address on the stack

3. **believing in & relying on an abstraction that is not 100% guaranteed & enforced**
   namely types and procedure interfaces in C

```
int f(float f, boolean b, char* buf);
```

Recurring problem: *mixing control & data*

In 1950s, Joe Engressia showed the telephone network could be hacked by phone phreaking, ie. whistling at right frequencies

http://www.youtube.com/watch?v=vVZm7I1CTBs

The root cause: in-band signaling

In 1970s, before founding Apple with Steve Jobs, Steve Wozniak sold Blue Boxes for phone phreaking at university

More countermeasures

We can take countermeasures against buffer overflows to prevent, migitate, or detect buffer overflows at different levels & different points in time, incl.

- at 'platform level' (as discussed last week)
  - invisible to the programmer
- in libraries
- testing (dynamic analysis) at runtime
  - aka DAST (Dynamic Application Security Testing)
- static analysis at or before compile time
  - aka SAST (Static Application Security Testing)

Generic defence mechanisms

- Reducing attack surface

  Not running or even installing certain software, or enabling all features by default, mitigates the threat
  - A particular instance of this is OS hardening

- Mitigating impact by reducing permissions

  Reducing OS permissions of software (or user) will restrict the damage that an attack can have
  - following the principle of least privilege

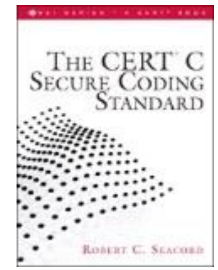
But: there will always be some high-privileged code that is an interesting target

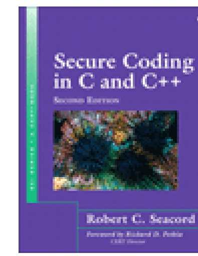
- eg `login` program will need access to the password file

Prevention

- Don't use C or C++
  - you can write insecure code in any programming language, but some make it easier…
- Better programmer awareness & training

Read – and make other people read – books like





  - CERT secure coding guidelines for C and C++ Online at www.securecoding.cert.org
  - Secure Coding in C and C++, R.C. Seacord
  - 24 deadly sins of software security, M. Howard, D LeBlanc & J. Viega, 2005
  - Secure programming for Linux and UNIX HOWTO, D. Wheeler
  - Building Secure Software, J. Viega & G. McGraw, 2002
  - Writing Secure Code, M. Howard & D. LeBlanc, 2002
  - …

Dangerous C system calls

Extreme risk

- `gets`

High risk

- `strcpy`
- `strcat`
- `sprintf`
- `scanf`
- `sscanf`
- `fscanf`
- `vfscanf`
- `vsscanf`
- `streadd`
- `strecpy`
- `strtrns`
- `realpath`
- `syslog`
- `getenv`
- `getopt`
- `getopt_long`
- `getpass`

Moderate risk

- `getchar`
- `fgetc`
- `getc`
- `read`
- `bcopy`

Low risk

- `fgets`
- `memcpy`
- `snprintf`
- `strccpy`
- `strcadd`
- `strncpy`
- `strncat`
- `vsnprintf`

[source: Building secure software, J. Viega & G. McGraw, 2002]

Better implementations of string libraries

- libsafe.h provides safer, modified versions of eg. strcpy
  - Prevent buffer overruns beyond current stack frame: Functions in this library check that they will not exceed stack frame

- libverify enhancement of libsafe
  - Functions in this library keep a copy of the stack return address on the heap, and checks if these match on returning

Like the platfom-level defences discussed last week, these are transparant to the programmer

- Program code hardly has to change (apart from importing a different library)

Better string libraries

- `strlcpy(dst,src,size)` and `strlcat(dst,src,size)` where `size` is the size of destination array `dst`, not the maximum length copied.
  - Less error-prone; consistently used in OpenBSD
- glib.h provides `Gstring` type for dynamically growing null-terminated strings in C
- Strsafe.h by Microsoft guarantees null-termination and always takes destination size as argument
- C++ string class

  C++ string objects are less error-prone than C strings
  - but `data()` and `c-str()` return a C string, ie. a `char*`, and result of `data()` is not always null-terminated on all platforms.

Safer dialects of C

Some approaches go further and propose safer dialects of C which include

- bounds checks
- type checks
- automated memory management with eg
  - garbage collection, or
  - region-based memory management

Examples: Cyclone, CCured, Vault, Control-C, Fail-Safe C, D, Rust

- Rust uses interesting combination of ownership and (im)mutability to avoid garbage collection

Runtime detection on instrumented binaries

There are many memory error detection tools that instrument binaries to allow runtime detection of memory errors, esp.

- out-of-bounds access
- use-after-free bugs on heap

with some overhead (time, memory space) but no false positives.

For example Valgrind (Memcheck), Dr. Memory, Purify, Insure++, BoundsChecker, Cachegrind, Intel Parallel Inspector, Discoverer, AddressSanitizer,...

Detecting out-of-bounds access requires additional administration in pointers, using so-called fat pointers

Fuzzing aka fuzz testing

A classic technique to find buffer weaknesses is fuzz testing

- **send *random, very long* inputs, to an application**
- **if there are buffer overflow weaknesses, this is likely to crash the application with a segmentation fault**

This is easy to automate!

More on fuzz testing in the security testing lecture.

Code review & static analysis

- **Code reviews**
  ie. someone reviewing the code manually
  Expensive & labour intensive

- **Code scanning tools** aka **static analysis**
  Automated tools that look for suspicious patterns in code; ranges for **CTRL-F** or `grep`, to advanced analyses

  Incl. free tools
  - RATS – also for PHP, Python, Perl
  - Flawfinder , ITS4, Deputy, Splint
  - PREfix, PREfast by Microsoft

  plus other commercial tools
  Coverity, PolySpace, Klocwork, Checkmarx…

Program verification

The most extreme form of static analysis:

program verification

> **proving by mathematical means (eg Hoare logic) that memory management of a program is safe**

- **extremely labour-intensive ☹**
- **eg hypervisor verification project by Microsoft & Verisoft:**
  - **http://www.microsoft.com/emic/verisoft.mspx**

Beware: in industry "verification" means testing,
in academia it means formal program verification

Conclusions

Moral of the story

- Don't use C(++), if you can avoid it
  but use a safer language that provides memory safety

- If you do have to use C(++), **become** or **hire** an expert

Required reading for this course

- Runtime countermeasures for code injection attacks against C and C++ programs by Yves Younan et al.
  - Not Section 3, 4.6 and all tables

- Sections 3.1 & 3.2 of lecture notes on language-based security

Exam questions: you should be able to

- Explain how simple buffer overflows work & what root causes are
- Spot a *simple* buffer overflow. format string attack, or integer overflow
- Explain how countermeasures - such as stack canaries, non-executable memory, ASLR, CFI, bounds checkers, pointer encryption, … - work
- Explain why they might not always work