Low level security

Andrew Ruef
What’s going on

- Stuff is getting hacked all the time
- We’re writing tons of software
  - Often with little regard to reliability let alone security
- The regulatory environment is pretty open
  - Due to our failures as technologists this might change
Why low level specifically?

- Programs written in C have two unfortunate intersecting properties
  - They do something important, servers, cryptography, etc.
  - By construction they permit subtle low level memory errors that allow for malicious attackers to completely compromise systems
How subtle are we talking?

- Heartbleed was undiscovered for two years
- Many bugs are found in released products even after internal pen testing and review
What is memory safety?

- A good question that started 2 days of discussion in the PL group
  - So probably no coherent answer yet

- Generally, memory safety assures *spatial* and *temporal* safety
  - Do not use memory after it is released
  - Do not write outside the bounds of an object
Could there be other errors?

- `goto fail` was not memory safety
- Some SSL CNAME checking errors were not memory safety
- Character conversion and “fail open” logic can still cause big problems
- Let’s just look at what we can find with tools for memory safety and correctness
What is clang-analyzer?

- A symbolic execution framework for C/C++ built in clang
- Operates on the clang AST
- clang-analyzer is actually separate from the LLVM project proper
- A core symbolic execution framework that drives state through compilation units
Extensible checkers

- A modular checker architecture where checkers “visit” state and
  - Do nothing
  - Create new state
  - Report a bug
Symbolic state

- The symbolic execution system keeps a symbolic state for every path it executes through a program.

- This state serves two purposes:
  - Checkers can query state to identify bugs.
  - When a bug is identified, the state is unrolled and projected onto the source code.
Extensible symbolic state

- Values stored in symbolic state are also extensible
- Checkers can define new types of values to store in the state
n2s(p, payload);
pl = p;

if (s->msg_callback)

1  Taking false branch →

s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
&s->s3->rrec.data[0], s->s3->rrec.length,
s, s->msg_callback_arg);

if (hbttype == TLS1_HB_REQUEST)

2  ← Assuming 'hbttype' is equal to 1 →

3  ← Taking true branch →

{
    unsigned char *buffer, *bp;
    int r;
Symbolic Execution

- Program testing technique
- Evaluate a program with symbolic variables instead of concrete variables
- Consider all branches and conditions that “might be” within a program
Example

```c
int nonneg(int a) {
    if(a >= 0) {
        return a;
    } else {
        return 0;
    }
}
```

How does a computer explore what this code does?

What if we could evaluate this program with every possible input?
Uses of symbolic execution

- This technique sits at the heart of modern flaw finding systems
- How could we use it as a tool?
What could we check?

- At each point in the program a checker visits, it has access to the current state.
- Values are symbolic; symbolic integer values include range.
- Some checkers that currently exist:
  - Array bounds
  - `malloc` size parameter overflow
  - Imbalanced mutex usage
Heartbleed

- Epic OpenSSL vulnerability that allowed for (somewhat) arbitrary read of heap data
- Ultimate cause – read object out of bounds
- Difficult to detect statically
- Could we write a checker to find it? How?
The bug

```c
int

ssl_process_heartbeat(SSL *s)
{
  unsigned char *p = &s->s3->rrec.data[0], *pl;
  unsigned short hbtype;
  unsigned int payload;
  unsigned int padding = 16; /* Use minimum padding */

  /* Read type and payload length first */
  hbtype = *p++;
  n2s(p, payload);
  p += 2;
  pl = p;
}
The bug

```c
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;

/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
```
Impact

- Read a specific amount of memory from the OpenSSL heap and send it to the client
  - Client is unauthenticated
  - Deliciously, the exfiltrated data sent to the attacker is encrypted
    - NIDS is useless, though you can see heartbeat messages with long sizes
Impact

- Deliciously, as long as the `memcpy` doesn’t produce a segmentation fault, this isn’t an observable attack in any systems security model.
  - HIDS and SELinux is useless.
How bad could this get?

- One SSL object allocated per connection
- Read values could include self or near-self referencing pointer values
- By establishing concurrent connections, could snapshot entire heap state

CloudFlare #heartbleed trivia: I dumped 43 GB of data, using 700k packets. One RSA prime is found 194 times. 0.02% prime to heartbeat ratio.
How could we find it statically?

- Need to know that `payload` variable is fully attacker controlled
- Use `ntohl` as an annotation that a variable is attacker controlled
- Identify unconstrained uses of those variables
What about the web?

- I want to live in a world with widespread verified code
- At the moment it is probably an easier sell to say that our medical and avionic systems should be formally verified
  - You mean they’re not right now?
  - Well, some of them are in Europe
- We can apply these same techniques to find bugs in web applications though
What are pragmatic things?

- There are some software security focused classes to take
  - Mike Hicks is teaching one in the fall on Coursera

- There are companies that will do you a good job on pen testing your stuff

- There are ways you can write and design your applications to make failure less certain
Programming Tips

- One way to view pointers is as capabilities*
  - A pointer is a language resource

- Every use of a pointer should be performed with concern to safety invariants
  - Ownership
  - Bounds
  - Lifetime

* http://www.pl-enthusiast.net/2014/07/21/memory-safety/
Ownership

- Which *thread of execution* is interacting with the pointer?

- Multiple threads interacting with shared memory is a source of both security and correctness errors

- Ownership questions usually resolved with a *mutex*
Bounds

- Is the access with the pointer *in bounds*
- Is the data being read/written within the bounds of the specified field or object?
- Bounds can sometimes be enforced via type checking
- For arbitrary buffers, carry around a size field and check the size before use
Lifetime

- Has an allocated pointer *fallen out of lifetime*, or *died*

- Using dead pointers is uncouth
  - Could result in writing into a now-live region of memory, resulting in *use after free*

- Control the lifetime of pointers with *reference counting*
Checking tools

- clang-analyzer is a static analyzer
- Dynamic analyzers can find bugs with fewer false positives and time spent
- Traces concrete execution of a program
- Examines the trace for violations of memory safety
Checking tools

- AddressSanitizer
  - Component of clang compiler
  - Emits code with checks embedded

- valgrind
  - Stand-alone checker, works on unmodified binaries
  - Executes code and checks for safety violations
Find Heartbleed with ASAN

- Fuzzer would need to produce heartbeat packets
- ASAN instruments reads and writes
- At runtime, the act of reading out of bounds triggers a fault
Avoid the snake

- Of course you should ask yourself, why am I writing in an unmanaged language?
- I think that Java and C# are on the front line of winning our war with memory safety
- So what could we do in the future and what is the frontier for making new programs better?
Types and memory safety?

- In some sense we already tolerate advanced static analysis of our programs before we let them run
  - We just call it “type checking”

- How much information about a program's behavior can we put into the type system?
  - Could we encode a state machine into the type system?
Neat type applications

- Session types
  - Essentially put state machine transitions into types

- Refinement types
  - Put logical constraints on the use of types
  - “Practical” implementations in LiquidHaskell, F7

- Check “high level” properties
  - Does “login” actually do the right thing?
We still have C though

- We *can* bolt a lot of checking onto C code though

- frama-c
  - Open source analysis framework
  - ACSL – specification language for behavior

- Write C code with low and high level guarantees
frama-c success stories

- Formally verified PolarSSL implementation
- After-the-fact retrofit of memory safety guarantees onto older C codebase
  - Helped by PolarSSL modularity
- Could we do better?
Compartmentalization

- ocaml-tls implements the TLS protocol in OCaml
- They use native code bindings to implement block ciphers
  - There are some reasons you want to do this like timing channels and performance
  - The code is small enough that you could formally verify it for memory safety
Well specified formats

- When sending or receiving rich data, let other coders worry about serialization and deserialization
- Encode messages into protocol buffers, CapnProto, thrift, etc.
Well encapsulated libraries

- If your crypto library abstraction wants to make you choose a cipher suite and decide if you want HMAC or not, you have a bad library.
- If your product depends on maintaining some kind of security invariant, consider hiring a security adult.
Conclusion

- It’s scary
- We have a lot of low level code
- We don’t know what it does
- We’re getting better
- Please don’t write more of it