Vulnerability Discovery

“Despair is when you’re debugging a kernel driver and you look at a memory dump and you see that a pointer has a value of 7”

Bugs

- Bugs are all over the place
- In security, we care about specific types of bugs
- We can broadly classify these bugs
  - Memory errors
  - Race conditions
  - “Logic Bugs”

Bugs

- Bugs are all over the place
- In security, we care about specific types of bugs
- We can broadly classify these bugs
  - Memory errors
  - Race conditions
  - Everything else

Memory bugs

- Often you will hear this abbreviated as “buffer overflow”
- What does this mean?
- Coercing a program to write outside the bounds of an allocated object
- “The write overflows” etc
- Actually some nuance here

Conceptualizing memory errors

- Memory errors have…
  - A destination buffer being written to
  - With some source value
  - Of some size
- The destination buffer can be
  - Stack allocated
  - Heap allocated
  - Global

Conceptualizing memory errors

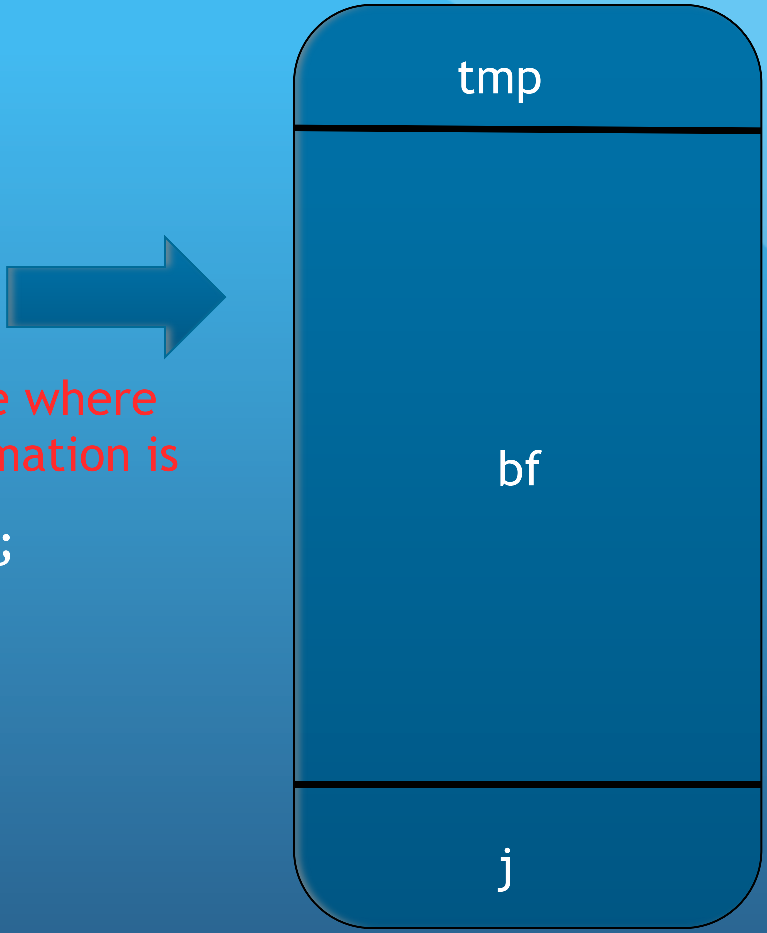
- The source can be
  - A static character
  - Input data
  - Non-input data
- The size can be
  - Fixed
  - Variable
  - Random

In pictures

```
void foo(int a, char *b) {
  int    tmp;
  char   bf[10];
  short  j;
  …

  memcpy(bf, b, strlen(b));
```

Only place where size information is

These do not match!



Finding the bugs

- There is actually a reasonable way to characterize “easy” vs. “hard” bugs
- Consider:
  - Bugs you care about will exist in reaction to input you provide
  - Read a program carefully, at each step ask yourself what decision it is making, and if it is because of your input
  - The further into a program you get from where your data is used, the harder the bug

Pattern for simple memory errors

- Look for where data is input into the program
  - `read/recv/recvfrom`
- Trace the use of that data
- Build a model for the data the program receives
- Start playing “what-if” games
  - What if this was <0 or >`MAX_SIGNED`

Do not be afraid of a dynamic study

- When you only have a binary, run it!
- Your debugger can help you answer some questions potentially faster than IDA
- “Where does the program input data?”
  - Breakpoint on recv
- “Where is the buffer that is read first used?”
  - Break-on-access on buffer passed to recv

Debugging and program understanding

- Doing vulnerability analysis on someone else’s code is almost identical to looking for bugs in code that you wrote
- “Yeah but I wrote the code, so I know what it’s doing”
  - How’s that working out for you?
- What tools do you use when you debug your own programs?
- Try and achieve understanding of what your target program is doing

Other kinds of memory errors

count comes from outside

- “Integer overflows”
  - Really these are logical errors which lead to memory errors

```
void foo(int handle) {
  int count,i;
  struct entry *entry_array;
  read(handle, &count, 4);
  entry_array = malloc(count*sizeof(struct entry));

  for(i = 0; i < count; i++) {
    entry_array[i].bar = …
```

Then, entry_array is much smaller than we think and this will walk into other memory

This multiplication could overflow and wrap around

Integer overflows

- In principle these are pretty easy
- Find locations where input-controlled (or input-influenced) variable ‘x’ is used to dereference into memory
- Search for example where some value of ‘x’ results in an out of bounds read or write
- Then you have found a vulnerability

Use-after-free

- Heap managers are tricky things
- In general, using a pointer value returned by `malloc()` after passing it to the `free()` function is very bad
- Exactly why will come later
- “use-after-free” is a pattern that could be re-stated as “use of object after lifetime expires”
  - As a rule I think we can agree this is pretty gauche

An example (from WebKit)

HTMLElement.cpp

```
[445] void HTMLElement::setOuterText(const String &text,
                                      ExceptionCode& ec)
…
[468]   RefPtr<Text> t = Text::create(document(), text);
[469]   ec = 0;
[470]   parent->replaceChild(t, this, ec);
…
[488]   Node* next = t->nextSibling();
[489]  if (next && next->isTextNode()) {
[490]     Text* textNext = static_cast<Text*>(next);
          /* Triggers Javascript Event */
[491]     t->appendData(textNext->data(), ec);
[492]     if (ec)return;
[493]     textNext->remove(ec);   ← Uses stale pointer
```

Finding these is harder

- You have to understand more of the program to find them
- You’re not just looking for states, but, conditions and paths
  - In the WebKit example did you see a `free()`?
- Look at the program and try and conceive of possible paths through a program where a value would be used after deletion

Side note: reference counting

- Memory management is hard on programmers
- Sometimes we try and bolt garbage collection into C
- One tactic is reference counting, attach an atomic integer to every object and increment it when some piece of code takes ownership of the object
- Frequently a source of bugs when someone forgets to increment a reference counter when they should
  - Frequently a source of impossible-to-find memory leaks when someone forgets to decrement BORING

Race conditions

- This gets fiendish fast
- Imagine all the fun of finding UAFs or heap corruption but with events happening at nondeterministic orderings
- Track events that happen in different threads, consider all possible orderings of those events
  - Use a debugger

Logic errors

- Trick the machine
- Some way to “convince” it that some security invariant has been met when it has not
- No general class, pattern, or scheme
- It’s not always about memory corruption or shell games
- If you can send the right sequence to a remote system that sends you a flag, that’s all that matters

A note about the frontier

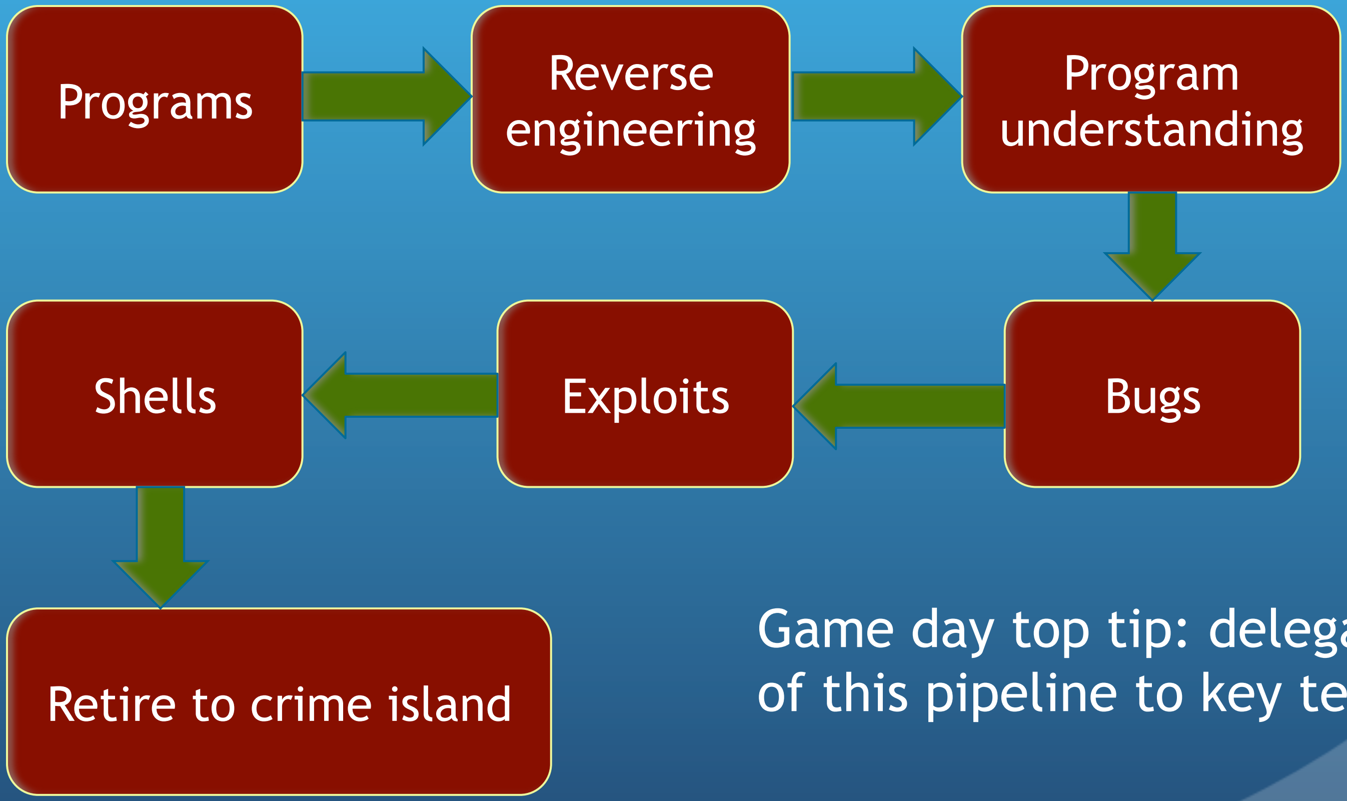
- “Where are we in finding all of these bugs automatically, in the real world”
- Some of our best technologies are dynamic
  - This is a way of saying “it sucks”
- Static systems have a lot of limitations
- For further reading, check out Regehr’s integer overflow checker, MS Research SLAM, Static Driver Verifier

Tools

- For binaries, you will want IDA
  - I think Hopper might be okay but I’m too committed to IDA to back out now
- You want help visualizing control flow
- You want to get the “shape” of a program into your head
- IDA is interactive
  - Rename things
  - Create comments

Reverse engineering

When working, your pipeline will look like this



Game day top tip: delegate stages of this pipeline to key teammates

Vulnerability mitigation

- When you identify a vulnerability, you can frequently identify how to change the code so the vulnerability is no longer there

- Can you apply these changes to the original program image?
  - Depends but almost always this answer is ‘yes’

- Binary patching – add extra invariants, conditionals

Binary patching

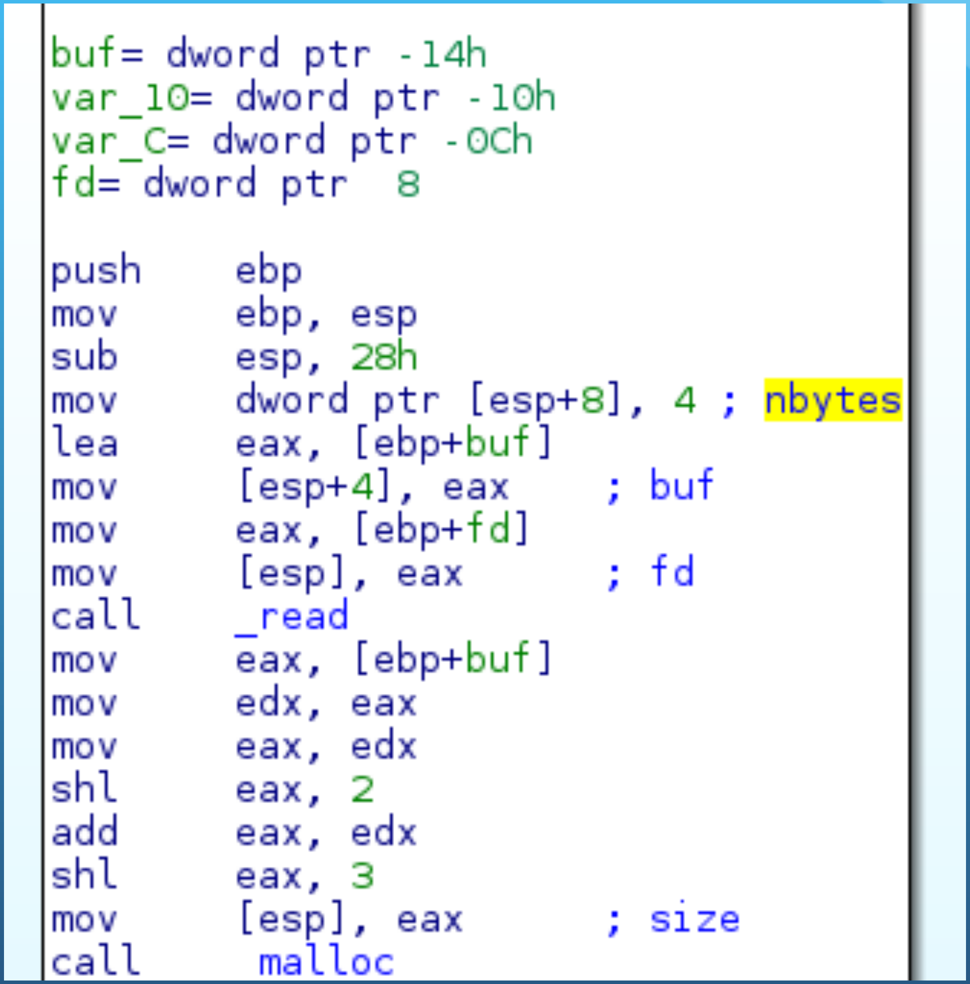
- In principle, simple
- Identify behavior you want to change
- Make the change in binary code
- Smush* your changes together with the original program image

An example, from before

- C code:

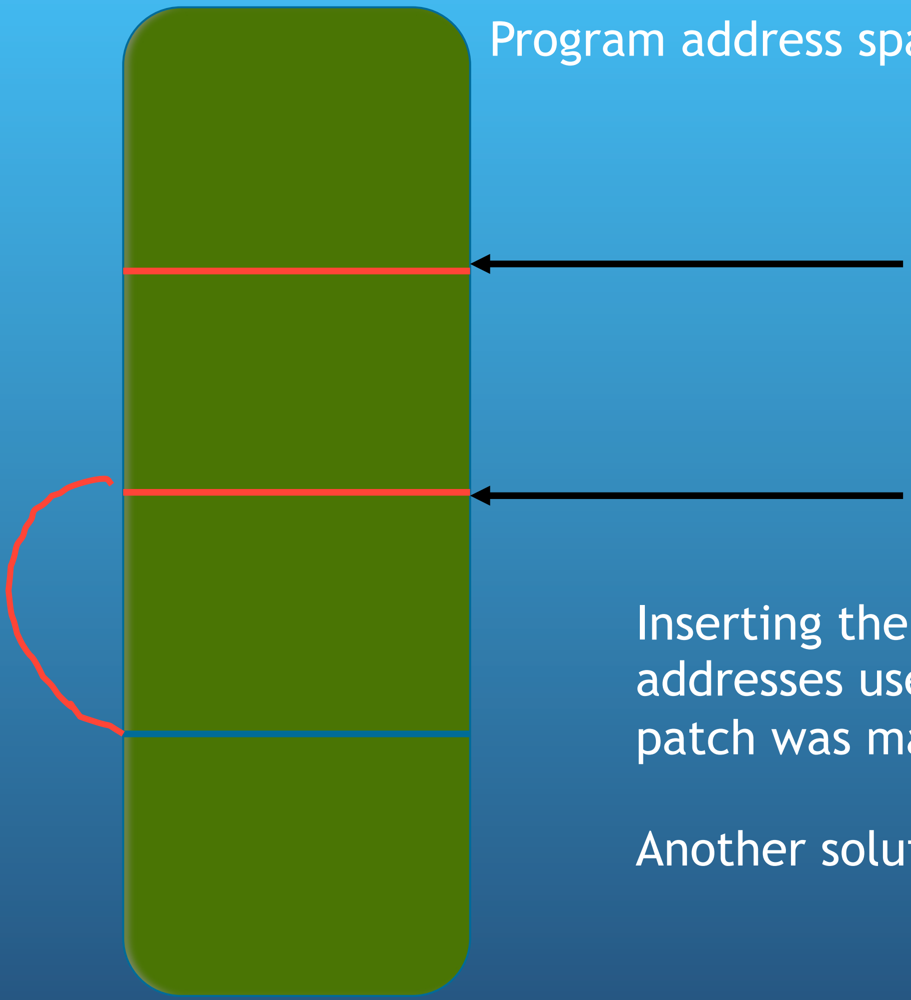
```
int foo(int handle) {
  struct entry  *entry_array;
  int   count,i;

  read(handle, &count, 4);

  entry_array = malloc(count * sizeof(struct entry));

  for(i = 0; i < count; i++) {
    entry_array[i].stuff[0] = 0;
  }

  return 0;
}
```

Binary code, from IDA



```
buf= dword ptr -14h
var_10= dword ptr -10h
var_C= dword ptr -0Ch
fd= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 28h
mov     dword ptr [esp+8], 4 ; nbytes
lea     eax, [ebp+buf]
mov     [esp+4], eax    ; buf
mov     eax, [ebp+fd]
mov     [esp], eax      ; fd
call    _read
mov     eax, [ebp+buf]
mov     edx, eax
mov     eax, edx
shl     eax, 2
add     eax, edx
shl     eax, 3
mov     [esp], eax      ; size
call     malloc
```

A direct approach



Program address space

Make code patch here

Other code here refers to addresses that would be nudged by insert

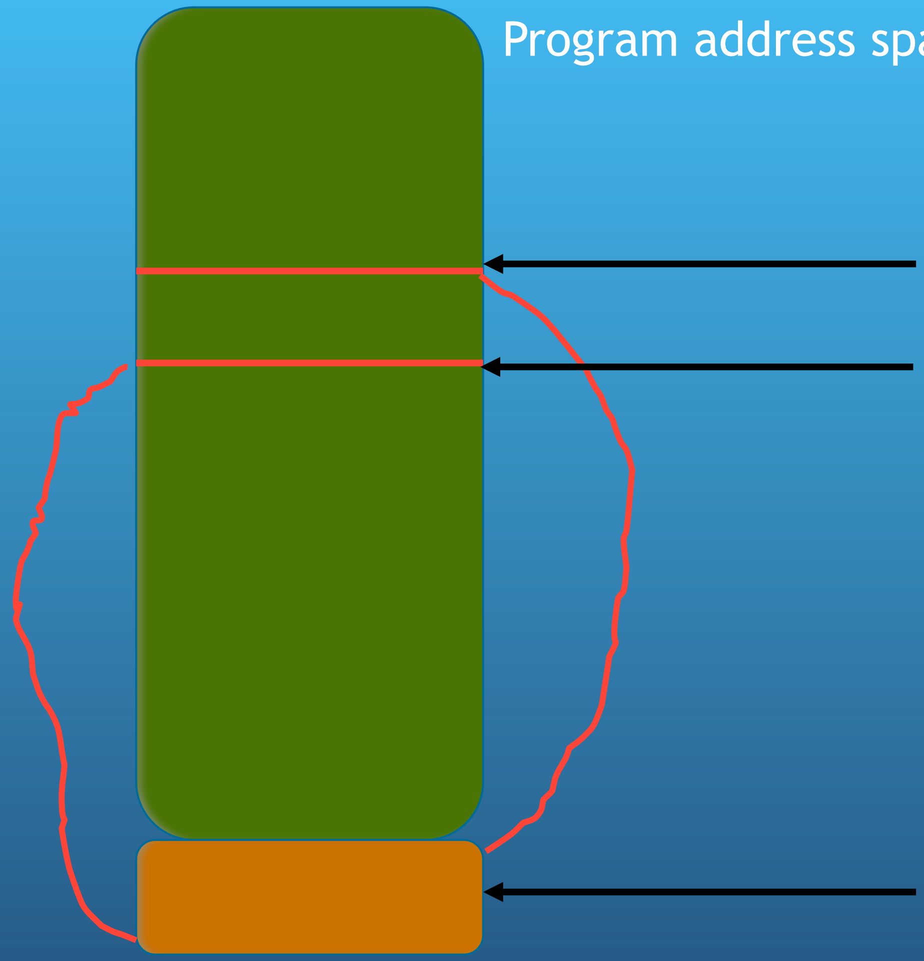
Inserting the code patch would destroy relative addresses used *after* the location where the patch was made

Another solution is needed…

Executable content editing

- Fortunately this is well traveled ground
  - The software piracy world has been modifying executable file formats since there were executable file formats
- Tools exist for adding code sections or expanding the sizes of existing sections
- Straightforward algorithm then emerges
  - Detour old code to new section
  - Have new logic
  - Jump back to end of old code that you removed

Illustration



Program address space

Beginning of vulnerable logic, detour to patch

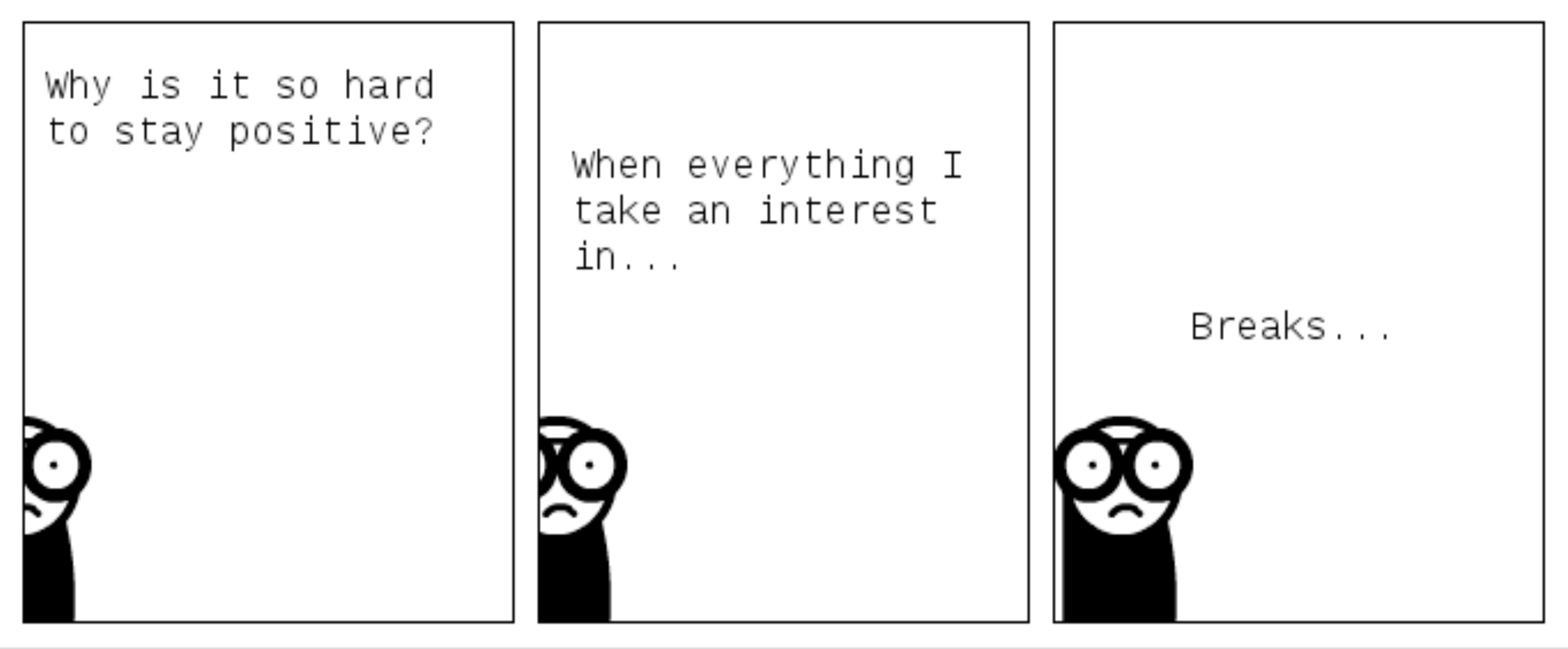
Rest of the program

Section with code effecting patch

Constraints on bugs (metagame)

- In CTF we don’t talk about real programs
- Some person wrote these programs to have specific bugs
- They (usually) try and scope it so that it is tractable for a weekends work
  - It is a “game”

Finding bugs is good for your life


Why is it so hard to stay positive?
When everything I take an interest in...
Breaks...

Finding bugs is good for your life

- Be a better programmer
- Be a better hacker
- Amaze your friends with ability to find small details that are wrong and could lead to total compromise