Exploitation and state machines

Programming the “weird machine”, revisited

Thomas Dullien / Halvar Flake

What is this talk about ?

- Keywords for conversations in the community: vulnerabilities, exploits, control-hijack, stability, write4 etc.
- “Clear to everybody"
- A lot of *folklore knowledge* -- impossible-to-attribute terms understood by people immersed in the community
- Occasionally, clashes with academia flare up when terms (“exploit”) are misappropriated
  - APEG - Hopping a few branches to reach vulnerability
  - AEG - Successfully performing a simple EIP hijack on a 2003-style example
  - Return-Oriented Programming - repeated chaining of existing code fragments

What is this talk about ?

- Reconsidering what we do:
  - What exactly is this thing we call “exploitation” ?
  - What are the current limits of automation ?
- Part 1 of the talk:
  - What *is* exploitation ?
  - What is the *right* way to think about exploitation ?
  - Why doesn't ASLR+DEP matter in many situations ?
- Tangentially related to this I will talk about
  - What is the role of the implicit state machines ?
  - What do these state machines mean for static analysis and automated input generation ?
  - Does existing theory capture any of it ?

Whose ideas are these ?

- The underlying ideas in the talk are clearly not "my" ideas
- Everybody that has built sophisticated exploits has parts of these ideas floating around in his brain
- TAOSSA mentions very similar ideas, less fleshed out
- Sergey Bratus seems to have coined the term “weird machine”
- Truth is: This is folklore knowledge that really should be put in writing somewhere (before we get 20 papers claiming invention)

What are programs ?

- Every instruction transforms a state into a new state
- “Traditional” way to look at code
- Is this the right way to look at things ? Isn't it overly fine-grained ?
- Slightly different viewpoint: Each interaction with a program transforms a state into a new state
- The programmer defines a set of “valid states” – a program can only do things allowed by these states, and executing attacker code isn't part of that

Memory corruptions ...

- So let's view programs as finite state machines
- Interaction causes transitions between states
- Assume all states are “under control” – e.g. no valid program state is insecure (for this presentation)
- … and then we corrupt memory ...
- Suddenly, the space of possible program states explodes in size

Weird machines ...

- The transition functions that map between states still exist
- They now they operate on invalid / absurd states
- With each interaction, we transform one invalid / absurd state into a new absurd / invalid state
- We have a new state machine now: One with gazillions of unknown states, and most transitions lead to instant death (crash)
- But in the end, this isn't much different from any CPU – at any point in time, most instructions will yield a crash
- Sergey Bratus called these things “weird machines”

So what *is* exploitation really ?

- Exploitation is setting up, instantiating, and programming the weird machine

So what *is* exploitation really ?

- The goal is to reach a state that violates security assumptions (ideally in the most egregious imaginable way)
- The “traditional” EIP-to-data hijack was just an easy way to transform the weird machine into one we understand well: Native CPU code
- In the end, we really do not care -how- we're performing computations inside the process address space

Weird machines – what are they ?

- Weird machines are application-dependent
- Weird machines are initial-state-dependent
- Weird machines are really hard to control – hence attackers spend a lot of time setting the initial state in a way that allows more controlled transitions
- Even then, probabilistic risk remains: Initial state is nondeterministic (unknown initial heap state due to inherent nondeterminism from multithreaded heap operations)

Examples for weird machine programs

- Bootstrapping regular executable code via chaining code chunks – short, transition to x86
- Mark Dowd's virtual shellcode work (patching out restrictions from the .NET/Java interpreter, executing unrestricted code)
- Fully chained payloads (Iozzo/Weinmann IPhone 2010)
- Peter Vreugdenhil ASCII/Unicode overflow-to-infoleak
- But much more: Pretty much any sophisticated heap exploit nowadays has a long set-up to start the weird machine in the right initial configuration

Consequences

- Mitigations fail routinely. The pattern is:
  - 10: Attackers use one “path” to program the weird machine
  - 20: Defenders mitigate against that path
  - 30: Attackers change the path slightly
  - 40: After a few months/years of ownage, a path becomes public. GOTO 20.

- Forensics on sophisticated exploits without the trigger can be very hard
  - Without network traffic, you might not have reproducibility
  - Your server did something bizarre and unexplainable – without understanding the initial state and the instructions of the weird machine, understanding is nigh-impossible

What does this mean for the attacker ?

- “Cut out all this handwaving – get to the meat”
- ASLR+DEP do not matter nearly as much as one would think
- Most of the time, they can be broken through clever weird machine programming
- Infoleaks are made, not found
- Sometimes, you can program the weird machine without an infoleak

Many ways that ASLR+DEP failed

- Phrack 58.4 (Nergal), Phrack 59.9 (Tyler Durden) (PaX specific, fixed)
- Dowd / Sotirov: How to Impress Girls with Browser Memory Protection Bypasses
- Dowd: Virtual Shellcode
- Blazakis: Btree element ordering pointer inference
- Blazakis: JIT spraying
- Currently en vogue: Programming the weird machine to create an info leak
    - Example: Peter Vreudgenhil's ASCII/Unicode Overlap
    - Dozens of others are floating around, standard fare for modern server-side attacks – every second researcher has a favourite
- In this talk (mostly as academic exercise): Hijacking the Spidermonkey Javascript Bytecode interpreter

Hijacking the Spidermonkey bytecode

- Spidermonkey compiles Javascript into a byte code
- This byte code is subsequently interpreted
- The byte code is trusted – it is assumed to be generated by the compiler, and not do anything evil
- The byte code is quite powerful – you can certainly do whatever you want once you execute arbitrary such code
- Adding values, copying data, etc. are all doable with a bit of effort

Useful instructions

- JSOP_POPN
- JSOP_DUP
- JSOP_DUP2
- JSOP_POPV
- JSOP_GOTOX, JSOP_GOTO
- JSOP_SWAP
- JSOP_ADD, JSOP_OR, JSOP_SUB etc.
- JSOP_IFEQ
- JSOP_SETLOCAL

Useful instructions

- JSOP_POPN

```
BEGIN_CASE(JSOP_POPN)
        regs.sp -= GET_UINT16(regs.pc);
END_CASE(JSOP_POPN)
```

Useful instructions

- JSOP_SWAP

```
BEGIN_CASE(JSOP_SWAP)
        rtmp = regs.sp[-1];
        regs.sp[-1] = regs.sp[-2];
        regs.sp[-2] = rtmp;
END_CASE(JSOP_SWAP)
```

Useful instructions

- JSOP_POPV

```
BEGIN_CASE(JSOP_POPV)
        ASSERT_NOT_THROWING(cx);
        fp->rval = POP_OPND();
END_CASE(JSOP_POPV)
```

Useful instructions

- JSOP_SETLOCAL

```
BEGIN_CASE(JSOP_SETLOCAL)
        slot = GET_UINT16(regs.pc);
        JS_ASSERT(slot < script->depth);
        vp = &fp->spbase[slot];
        GC_POKE(cx, *vp);
        *vp = FETCH_OPND(-1);
END_CASE(JSOP_SETLOCAL)
```

Amusing complications

Some amusing complications arise due to Spidermonkey's peculiar handling of values that double as pointers to objects.

Writing large values can be challenging, and writing to non-dword aligned locations takes a bit of imagination.

It's fun, though, and much less annoying than writing code in JITsprayed operands.

Slightly amusing scenario

- Ok, let's assume we have a semi-controlled write
  - We can write a value that we control
  - The destination address is not really controlled – a random 24 bit value will be added before the write
  - The write will be DWORD aligned though – so only 22 bit of random

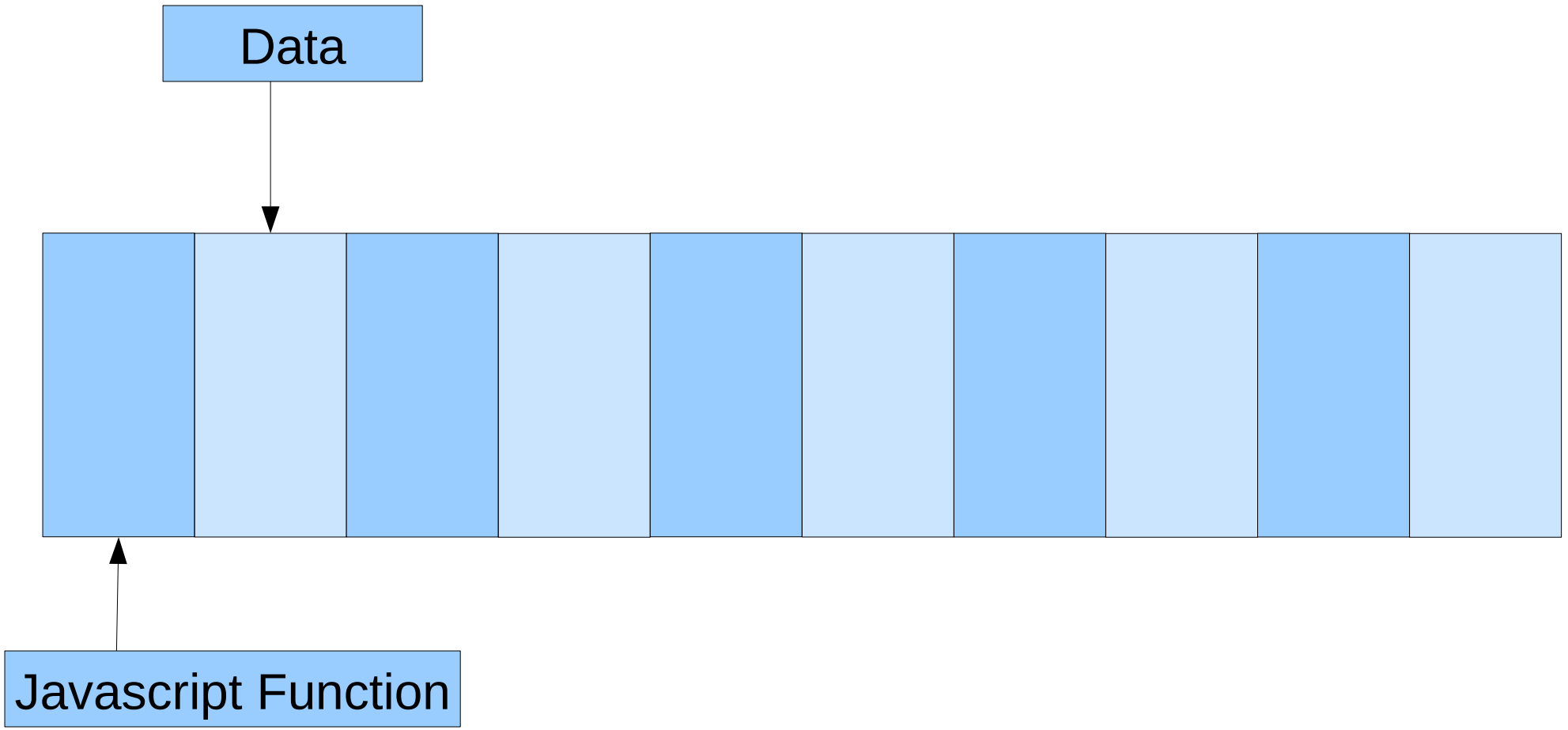
- ASLR + DEP are enabled – we don't know any code addresses, but we vaguely know (with some margin of error) where the heap is at

- Can we get reliable execution ?

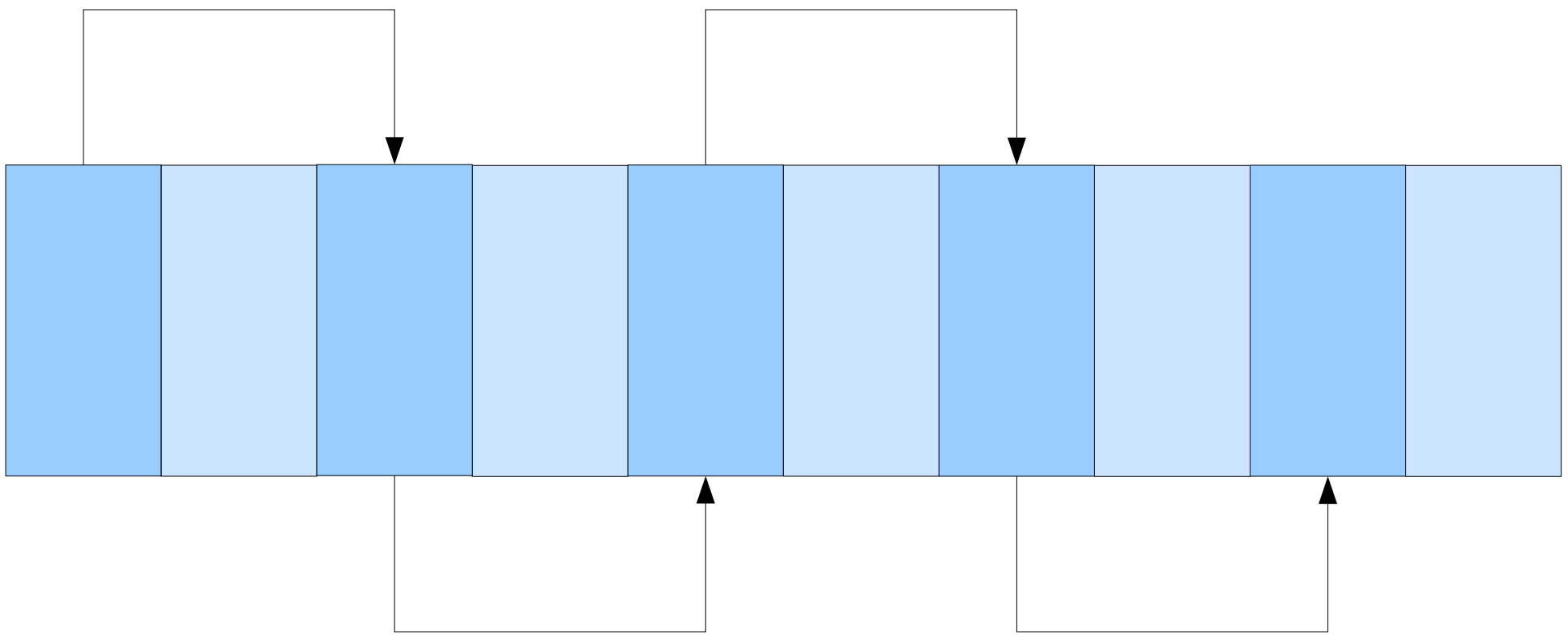
Battle plan

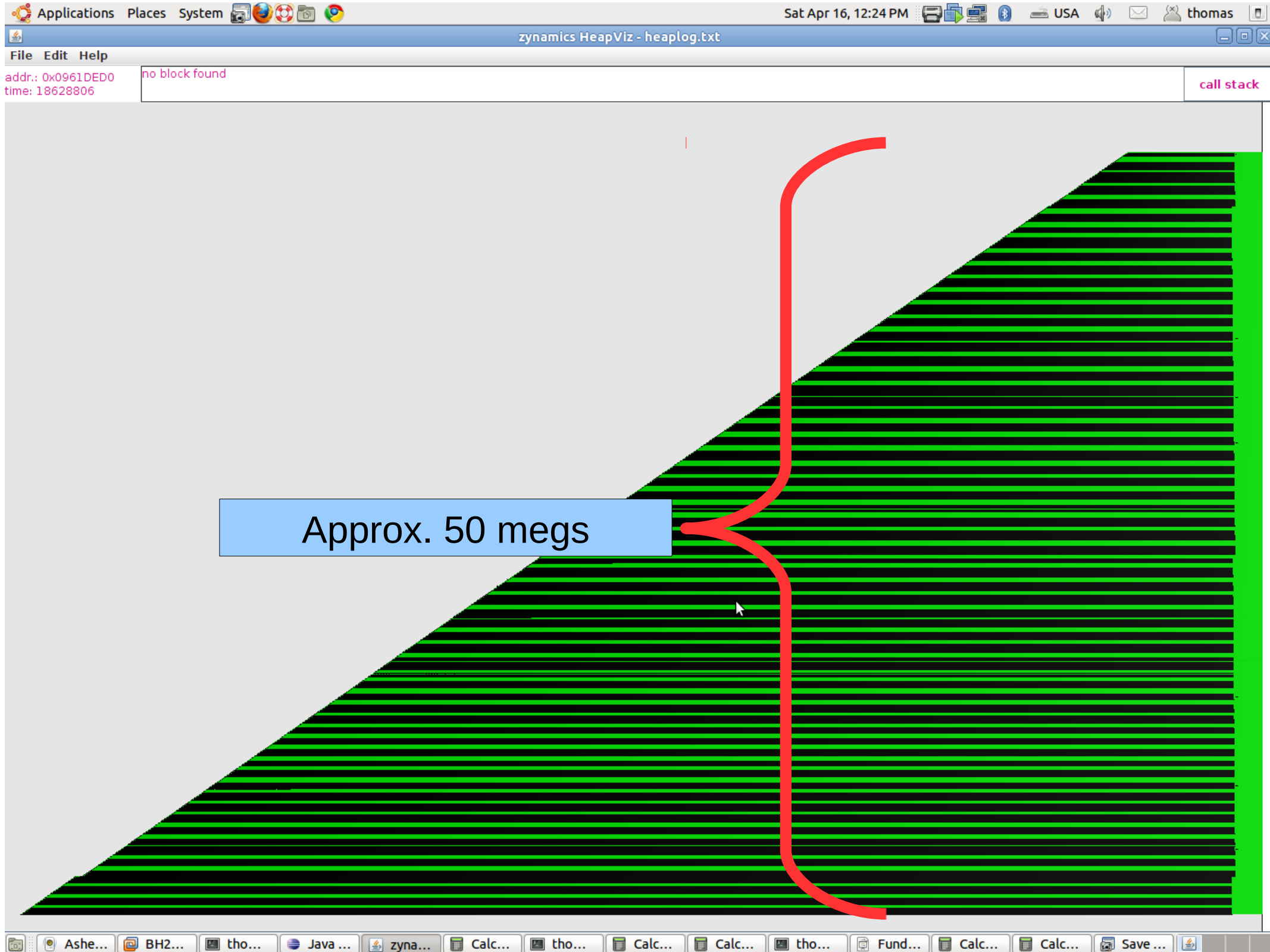
- Fill the heap with the bytecode of Javascript Functions (generated by the Javascript compiler)
- Make sure these functions “call each other”
  - Each function calls the “next” function in the chain
- Interleave those bytecode arrays on the heap with controlled data
- Write a jump jump into bytecode stream
- Hijack execution of the bytecode interpreter

Diagram



Diagram




Applications
Places
System
Sat Apr 16, 12:24 PM
USA
thomas
zynamics HeapViz - heaplog.txt
File
Edit
Help
addr.: 0x0961DED0
time: 18628806
no block found
call stack
Approx. 50 megs
Ashe...
BH2...
tho...
Java ...
zyna...
Calc...
tho...
Calc...
Calc...
tho...
Fund...
Calc...
Calc...
Save ...

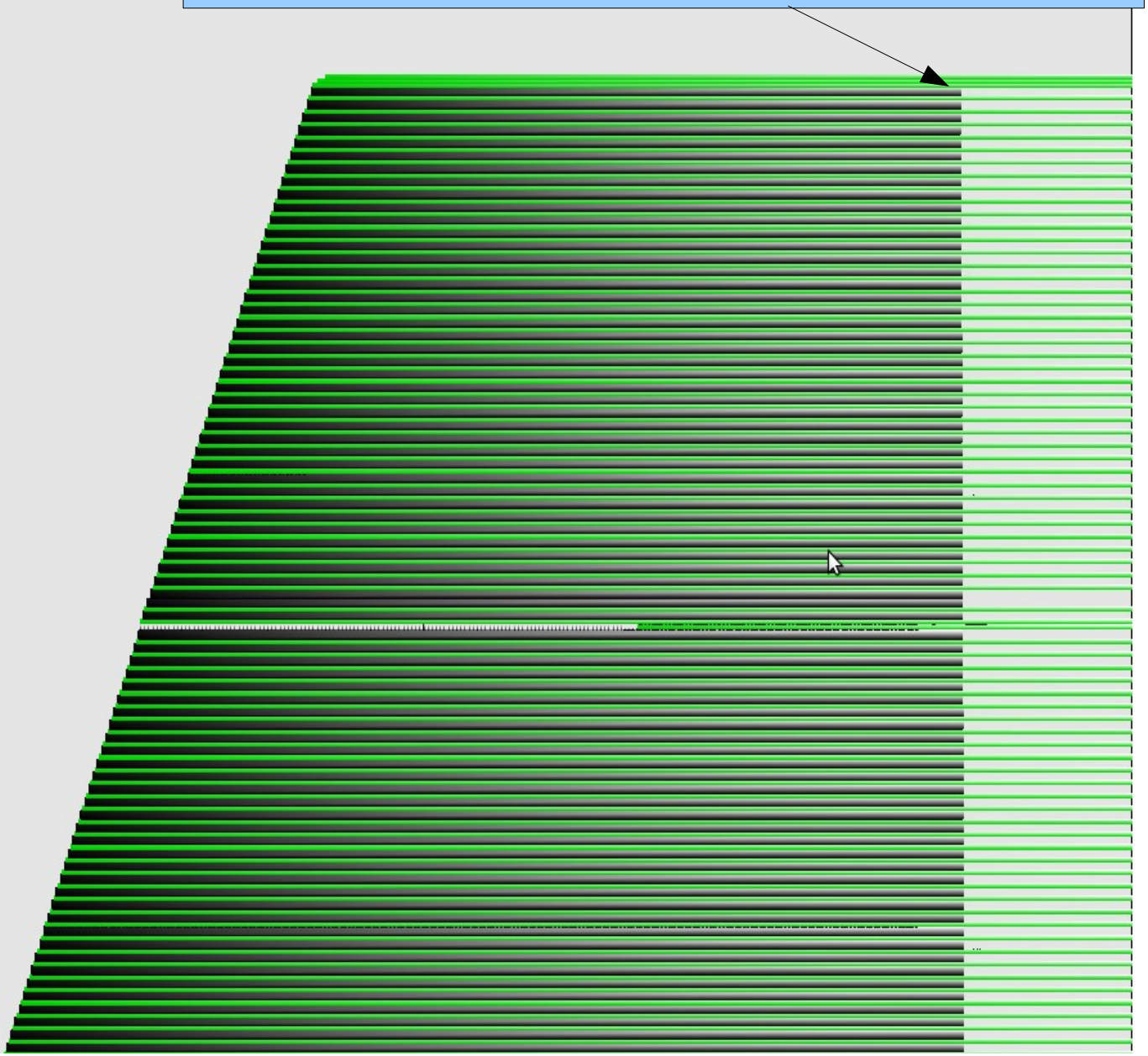
File Edit Help

addr.: 0x0B146749
time: 27398892

no block found

call stack

Garbage collection kicks in,
just for illustration



Data is black

Bytecode is
green

Hijacking the stream

- What Javascript are we going to populate memory with ?

  f(v,idx,arr){ v++(...);if(idx < MAX) arr[idx+1](v,idx+1,arr); }

- Streams of “v++;” get compiled to

  0x63 0x00 0x00 arginc 0

  0x51 pop

- 4-byte aligned, for your convenience
- Overwrite with JSOP_GOTO 0x06 0xXXXX
- Alternatively, with JSOP_GOTO 0x8B 0xXXXXXXXX

Summary

- Extremely simplified scenario
- Within Adobe Reader, semi-controlled write4 with a 25-megabyte margin-of-error will still be moderately easily exploitable in the presence of ASLR+DEP
- This should hold for IE and Firefox, too

Questions for part 1 ?

No, we don't need the interpreter

- Having an interpreter certainly makes life easy
- There's many of these – think Postscript, Glyph Scaling, Javascript, Flash, Python, PHP etc. etc. etc.
- Exercise: Analyze Microsoft's Javascript implementation for the same sort of fun
- But you do not need the interpreter
- The take-away from this presentation is: You *can* and *should be* programming the weird machines
- Obsession with taking EIP might sometimes be a bad thing

Credit for prior art

- Sergey Bratus for his RSS talk 2009

  http://www.cs.dartmouth.edu/~sergey/hc/rss-hacker-research.pdf

- The TAOSSA Crew (Justin Schuh, Mark Dowd, John McDonald) for having implicitly formulated a lot of this

- Stefan Esser for prior work on the PHP interpreter (Syscan 2010)

- Mark Dowd for his contributions on Virtual Shellcode (PacSec 2010)

- I am sure I have forgotten more, please ping me so you get added

Part 2: Implicit state machines

- This part is less constructive than the first part
- It will mostly be a critical examination of the actual capabilities of the tools & theory available to us
- I don't have solutions
- I'll just discuss a bunch of things that we (as a species) really don't know how to do yet

Implicit state machines

- A lot of folks want to do “automated exploitation”
- Bugfinding → input crafting to reach vulnerability → Execution Hijack
- Static Analysis → Dynamic symbolic execution → SMT Solving
- A common mistake made by *everybody*: Viability of a code path depends heavily on application state
- Getting an application into the right state can be hard
- There is always an implicit state machine in the way
- Ignoring the implicit state machines leads to failure (or worse, overstating your actual achievement and having that published)

Why is input crafting hard ?

- Many folks have gone down the route “generate constraints from program path, throw into solver, pray for result”
- How do you treat global state here ?
- What if the program wants you to issue a “EHLO” first ?
  - The necessary state modifications will happen on a different program path
- If you don't know sets of possible values for global state variables, you won't know if a path is feasible or not
- Real software almost always requires you to put the program into a state that will permit the vulnerable code path. This is a recursive problem: To put the software into that state, you might to have to first put it into another state.

  Ignoring the implicit state machine is not helping.

Determining nonexploitability is hard

- A crash is always a symptom – the root cause is something else
- The root cause is usually -semantic- in nature – not checking something, misunderstanding something, misusing something
- The root cause and the crash can be arbitrarily far removed – a process can crash hours after the actual root cause for the crash
- In order to determine exploitability, one would have to:
  - Backtrack to find the -semantic- root cause
  - Explore program states forward from there to see what one can do
- That stuff is hard.
- !exploitable pays for trying to solve the problem cheaply by being essentially a biased coinflip

Why didn't static analysis kill all bugs?

- Ask a static analysis guru:
  - Why have your tools not killed all bugs (or found really awesome bugs) ?
- Usual answers are:
  - Interprocedural analysis is hard
  - C++ analysis is hard
  - “They will, next year”
  - “They aren't used widely enough”
- While most of the above is true, there are other reasons, too

Why didn't static analysis kill all bugs?

- Let's check a real bug & reduce it: crackaddr() overflow
- Reduced to small example
  - 60 lines of C code,
  - one function
  - simple stack overflow
  - yet no static analyzer can distinguish vulnerable from non-vulnerable

File Edit Search View Tools Options Language Buffers Help

simple_example_bad.c

```
int copy_it( char * input ){
    char localbuf[ BUFFERSIZE ];
    char c, *p = input, *d = &localbuf[0];
    char *upperlimit = &localbuf[ BUFFERSIZE-10 ];
    int quotation = FALSE;
    int roundquote = FALSE;

    memset( localbuf, 0, BUFFERSIZE );
    while( (c = *p++) != '\0' ){
        if(( c == '<' ) && (!quotation)){
            quotation = TRUE;
            upperlimit--;}
        if(( c == '>' ) && (quotation)){
            quotation = FALSE;
            upperlimit++;}
        if(( c == '(' ) && ( !quotation ) && !roundquote){
            roundquote = TRUE;
            /*upperlimit--;*/}
        if(( c == ')' ) && ( !quotation ) && roundquote){
            roundquote = FALSE;
            upperlimit++;}
        // If there is sufficient space in the buffer, write the character.
        if( d < upperlimit )
            *d++ = c;
    }
    if( roundquote )
        *d++ = ')';
    if( quotation )
        *d++ = '>';

    printf("%d: %s\n", (int)strlen(localbuf), localbuf);
}
```

li=81 co=37 INS (CR+LF)

File Edit Search View Tools Options Language Buffers Help

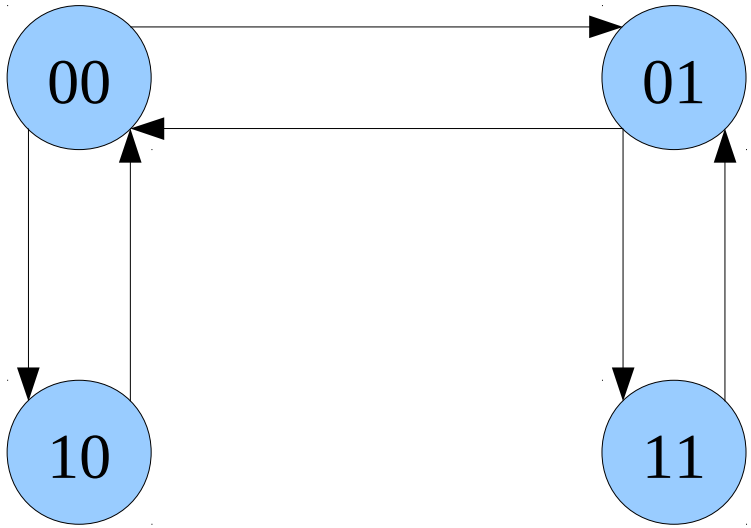
simple_example_bad.c

```
int copy_it( char * input ){
    char localbuf[ BUFFERSIZE ];
    char c, *p = input, *d = &localbuf[0];
    char *upperlimit = &localbuf[ BUFFERSIZE-10 ];
    int quotation = FALSE;
    int roundquote = FALSE;

    memset( localbuf, 0, BUFFERSIZE );
    while( (c = *p++) != '\0' ){
        if(( c == '<' ) && (!quotation)){
            quotation = TRUE;
            upperlimit--;}
        if(( c == '>' ) && (quotation)){
            quotation = FALSE;
            upperlimit++;}
        if(( c == '(' ) && ( !quotation ) && !roundquote){
            roundquote = TRUE;
            /*upperlimit--;*/}
        if(( c == ')' ) && ( !quotation ) && roundquote){
            roundquote = FALSE;
            upperlimit++;}
        // If there is sufficient space in the buffer, write the character.
        if( d < upperlimit )
            *d++ = c;
    }
    if( roundquote )
        *d++ = ')';
    if( quotation )
        *d++ = '>';

    printf("%d: %s\n", (int)strlen(localbuf), localbuf);
}
```

Remove comment to fix bug

li=81 co=37 INS (CR+LF)

The state machine



- We need to cycle through 00 → 01 → 00 → (…) many times to push the upper pointer outside of bounds
- We need to then perform at least 100 iterations to copy data
- Then we have a standard stack smash

Why does static analysis fail ?

- Most abstract interpretation-style analyses will try to map program lines to sets of states for variables
- Some of the more sophisticated analyses use relational domains (e.g. putting multiple variables into relationship to each other)
- They tend to “combine” different states using something like a union operator

Why does static analysis fail here ?

- When control flow converges, states are merged and “safely approximated”
- So “state 00 and p between 0 and 4” combined with “state 01 and p between 0 and 2” will be combined into “state 00 or 01 and p between 0 and 4”
- This contains spurious states: 01 and p=4 can't actually happen
- More precision is lost on each iteration of the loop
- So … uhm … even when we could solve all the interprocedural analysis and C++ issues, we still fail on heavily simplified versions of real-world code

Summary

- Automated input crafting that ignores the implicit state machine is useless in most real-world scenarios
- Determining that a bug identified by a crash is not exploitable is impossible without a full program trace up until that point – and even then it's totally unclear how you'd go about it
- Static analysis is powerful for large classes of bugs – but memory-copying loops with multiple internal states still mean failure most of the time
- Clearly, one can construct & find examples where all of this works – but just because I can construct one example where I do not fail does not mean I succeed.