Are Reverse Engineering and Exploit Writing an Art or a Science?

NYUPoly THREADS conference Panel
Cyber Security Awareness Week

November 14th 2013, Brooklyn, NY, USA

with **Dion Blazakis, Sergey Bratus, Dan Caselden, Brandon Edwards, Travis Goodspeed, Pete Markowsky, Meredith Patterson** and **Chris Rohlf**
moderated by **Julien Vanegue**

Introduction

Reverse Engineering and Exploit Development can be performed on programs in order to **Discover** and **Circumvent**:

- Data protection schemes.
- Exploit prevention (“mitigations”) schemes

Common involved procedures can be:

- Static (Manual or Automated analysis)
- Runtime (Debugging, Fuzzing, Tracing, …)

Foundations

"Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning."

"The most important property of a program is whether it accomplishes the intentions of its user."

-- Sir C.A.R. Hoare, "An Axiomatic Basis for Computer Programming", 1969

Art

The end is where we start from. And every phrase

And sentence that is right (where every word is at home,

Taking its place to support the others,

The word neither diffident nor ostentatious,

An easy commerce of the old and the new,

The common word exact without vulgarity,

The formal word precise but not pedantic,

The complete consort dancing together)

Every phrase and every sentence is an end and a beginning,

Every poem an epitaph.

-- T.S. Eliot, “Little Gidding”, 1942

Science

- An expression of the **Null hypothesis** for computer security can be : “The program will accomplish its user’s intentions”.

- Across this presentation, first class citizens are **Exploits**. An exploit is a *proof by construction* that disproves the null hypothesis.

Part 1

The Art and Science of Reverse Engineering

Semi-Automated Keygen Generation

- XOR Algorithm II crackme by ksydfius
  - Crackme uses an XOR + offset scheme
- Created an automatic keygen using:
  - IDA Pro disassembler by Ilfak Guilfanov
  - A symbolic emulator to generate SMT formulae from user defined traces (Used the theory of Arrays to model the strings and output)

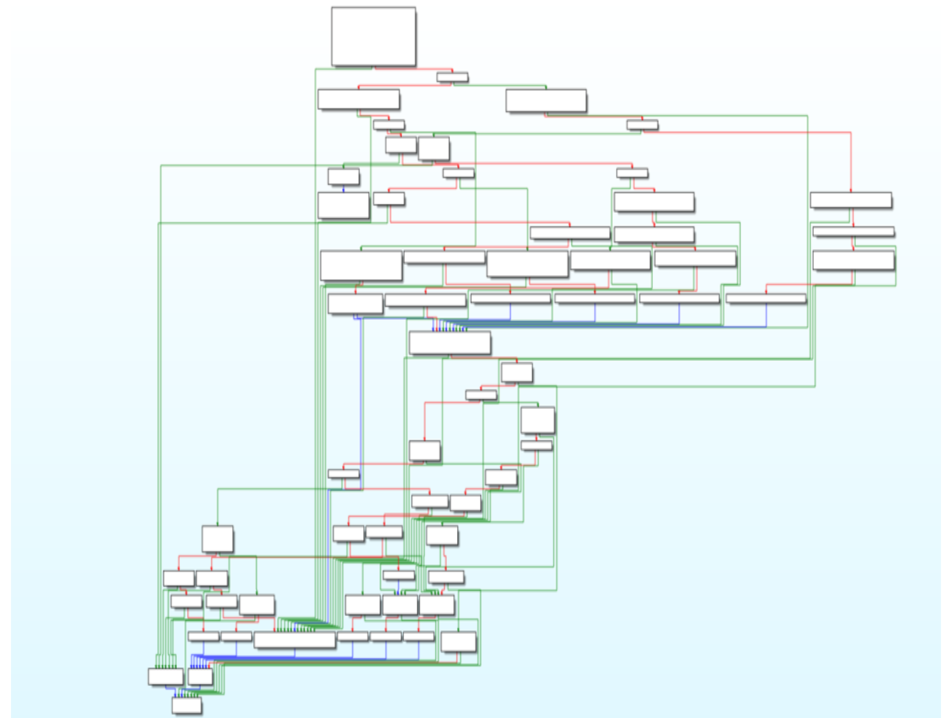
SAKG : example

```
.text:0040106E encode_user_serial proc near          ; CODE XREF: DialogFunc+2C↓p
.text:0040106E                 push    ebp
.text:0040106F                 mov     ebp, esp
.text:00401071                 sub     esp, 10h
.text:00401074                 xor     ebx, ebx
.text:00401076                 xor     ecx, ecx
.text:00401078                 xor     edx, edx
.text:0040107A                 cmp     eax, 20h        ; user_input must be 32-chars
.text:0040107D                 jnz     short loc_4010B0
.text:0040107F                 lea     edi, lookup_table ; "We go about our daily lives understandi"...
.text:00401085                 call    find_starting_point ; edx := sum of the first 32 chars of user input
.text:0040108A                 mov     ebx, edx
.text:0040108C                 call    mod_by_20_hex   ; ebx := 0, edx := (sum of first 32 chars) % 32
.text:00401091
.text:00401091 loc_401091:                             ; CODE XREF: encode_user_serial+40↓j
.text:00401091                 lea     esi, user_input_String
.text:00401097                 cmp     byte ptr [edi], 0
.text:0040109A                 jz      short loc_4010B0
.text:0040109C                 add     esi, edx
.text:0040109E                 mov     al, [esi]       ; move char from user_input_string to al
.text:004010A0                 xor     [edi], al       ; lookup_table[i] := user_input[idx] ^ lookup_table[i]
.text:004010A2                 add     [edi], dl
.text:004010A4                 add     bl, [edi]       ; mod_by_20_hex zeroes ebx so this is always zero
.text:004010A6                 add     bl, dl          ; bl := (xor-ed(char) + dl) + dl
.text:004010A8                 call    mod_by_20_hex
.text:004010AD                 inc     edi
.text:004010AE                 jmp     short loc_401091
.text:004010B0 ; ---------------------------------------------------------------------------
.text:004010B0
.text:004010B0 loc_4010B0:                             ; CODE XREF: encode_user_serial+F↑j
.text:004010B0                                         ; encode_user_serial+2C↑j
.text:004010B0                 add     esp, 10h
.text:004010B3                 pop     ebp
.text:004010B4                 retn
.text:004010B4 encode_user_serial endp
```

```
41  def Main():
42      solver = z3.Solver()
43      user_input_array = z3.Array("input", z3.BitVecSort(8), z3.BitVecSort(8))
44      output_str = z3.Array("output_str", z3.BitVecSort(8), z3.BitVecSort(8))
45      solver, lookup_table = CreateLookupTable(solver, lookup_table_str)
46
47      # add the path conditions that the user_str_index is equal to the sum of the
48      # first 32 characters
49      user_str_idx = FindStartingPoint(user_input_array)
50      # make sure the user_str_idx stays modulo 32
51      user_str_idx = user_str_idx % 0x20
52
53      # create conditions from the encoding loop
54      for i in xrange(240):
55          output_str = z3.Store(output_str, i, (user_input_array[dl] ^ lookup_table[i]) + user_str_idx)
56          user_str_idx = (output_str[i] + user_str_idx) % 0x20
57
58      # limit input to the printable ascii range
59      for i in xrange(0x20):
60          solver.add(z3.And(z3.UGE(user_input_array[i], 32),
61                           z3.ULT(user_input_array[i], 127)))
62
63      # Make sure our output_string is constrained to match our expected result
64      for i in xrange(len(encoded_str)):
65          solver.add(output_str[i] == ord(encoded_str[i]))
66
67      if solver.check() == z3.sat:
68          print 'SAT'
69          m = solver.model()
70          print 'Valid Serial is: ' + MakeInputStr(str(m[user_input_array]))
71      else:
72          print 'UNSAT'
```

Automatically Identifying All Valid Inputs to a Port Verification Function

- Needed to determine the list of valid ports
  - Fed to the function as memory address
- Function properties
  - Lots of arithmetic checks
  - No loops!
  - Jump table



Collect ALL the ports!

- Process
  - Convert each path from start to end to an SMT formulae and take the disjunction of the resulting path conditions.
  - While the formulae is SAT
    - Get a model for the port number and record the value
    - Add a new constraint specifying that the port number != the number we just found.
  - ROI (complete in 30 seconds vs more than 2 hours by hand)

```
ports = []
# The port is 2 bytes and we reprsent memory as bytes
# This is what the memory is in the location that the port number to be
#  validated exists at when the function begins
read_short = z3_wrapper.Concat(ents[10005], ents[10004])
final_constraints = [] # Accumulated previous valid ports to !
for ents in terms:
    while True:
        # Get a model for a valid port (R0 == 0 at end of function)
        # sat will == False when no more valid ports are possible
        solver, sat, model = ents.ObjSolve(final_constraints+[ents.R0 == 0])
        if not sat:
            break
        # Evaluate what the port value would be in this satisfiable model
        port_num = model.eval(read_short).as_long()
        print("PORT:", hex(port_num))
        ports.append(port_num)
        # Accumulate our valid ports so that next time around we will get
        #  a different possible solution.
        final_constraints.append(read_short != port_num)
```

Scalability of Symbolic Execution

- Symbolic Execution has trouble reasoning about full systems.
- What happens to vanilla symbolic execution?
  – Often gets stuck exploring argument parsing and error reporting forever.
- One classic issue is ‘Path Explosion’

Example of Path Explosion

```
num_dots = 0
for c in sys.argv[1]:
        if c == '.':  num_dots += 1
if num_dots == 1:
        assert("Nooooo!")
```

Even for simple cases, the program has a large number of paths. For example, see the simple cases where:

```
len(sys.argv[1]) = 1
len(sys.argv[1]) = 2
len(sys.argv[1]) = 3
```

Workarounds

- Most successful tools:
  - Mark less data ‘symbolic’
  - Explore fewer paths
- Methods:
  - Mix concrete and symbolic execution
    - Use snapshots or inputs as checkpoints
    - Start with sample input and build constraints dynamically
  - Identify & prune uninteresting or redundant paths
  - Selective symbolic execution
    - Skip libraries, kernel
    - Just explore this function locally
  - Many, many more.

Part 2

The Art and Science of Vulnerability Discovery

WebKit - Use After Free

- WebKit has had many vulnerabilities in its DOM/WebCore code
- Everyone treats them like magic when found with fuzzers
  - Root cause analysis shows the pattern is simple/easy to extract
  - Once you understand the pattern, apply it to other DOM code
- Code pattern is simple if you understand underlying components:
  - Reference Counting (RefPtr, PassRefPtr, OwnPtr) templates
  - JavaScript events
  - TCMalloc overloaded new/delete
    - WebKit has no garbage collector, Javascript engine does
- Root cause analysis -> apply the pattern -> find more bugs
  - Finds exploitable bugs in a focused way
  - No complex reasoning required

WebKit - Use After Free

- When might the code not be able to guarantee the lifetime of the object?

  Example: Javascript Event Callbacks

- The RefPtr documentation actually tells you what WebKit UAF looks like.

"If ownership and lifetime are guaranteed, a local variable can be a raw pointer but if the code needs to hold ownership or guarantee lifetime, a local variable should be a RefPtr"

WebKit - Use After Free

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);
/* The call below triggers JS event that will remove node pointed by textNext */
[491]     t->appendData(textNext->data(), ec);
[492]     if (ec) return;
[493]     textNext->remove(ec);  ← Uses stale pointer
```

Part 3

The Art and Science of Exploit generation

Exploit Development as Art

- **Application Specific Attacks**
  - Usually require detailed and "cross layer" understanding of an application's semantics
  - What humans do best? (moving between layers of abstraction?)
- **Mark Dowd's ActionScript:**
  - Confusion between x86 and bytecode execution
- **Comex's star exploit :**
  - Font program for runtime calculation (see Sogeti write-up)
- **Exploit primitive alchemy** :
  - Pivoting to better control

Exploit Development as Science

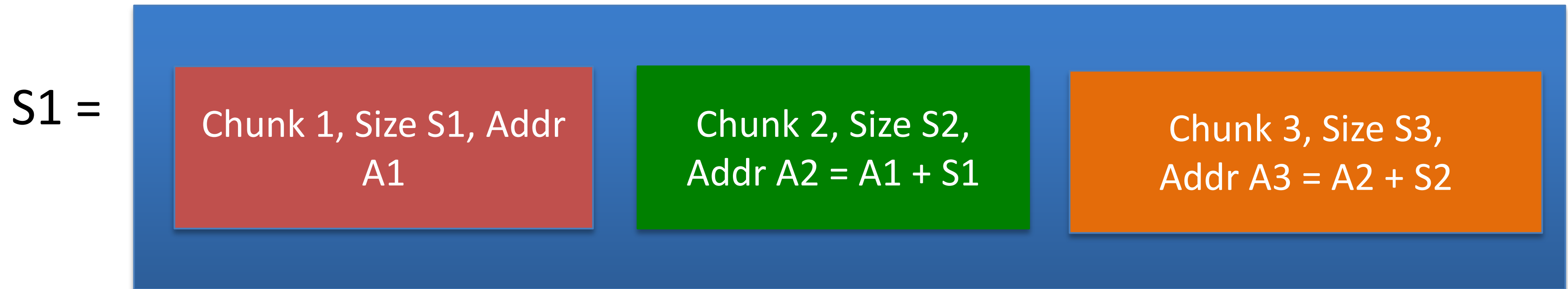
- **What we just talked about: Vulnerability discovery**
  - Everyone has a plan for this – AI, SMT, SE, GP/GA, ….
  - See co-panelist's thoughts on machine-assisted auditing
- **ROP-chains**
  - Could be more important in a future with runtime diversification (fine grained ASLR) – ROP computed at exploit time
- **Controlled/influenced values**
  - Dataflow analysis can tell us about these
- **Fuzzing for primitives**
  - Instrumentation + a search algorithm for discovering states with complex constraints

A common problem: non-determinism in programs

Assume attacker can overflow chunk 1 and chunk 3 is a target:

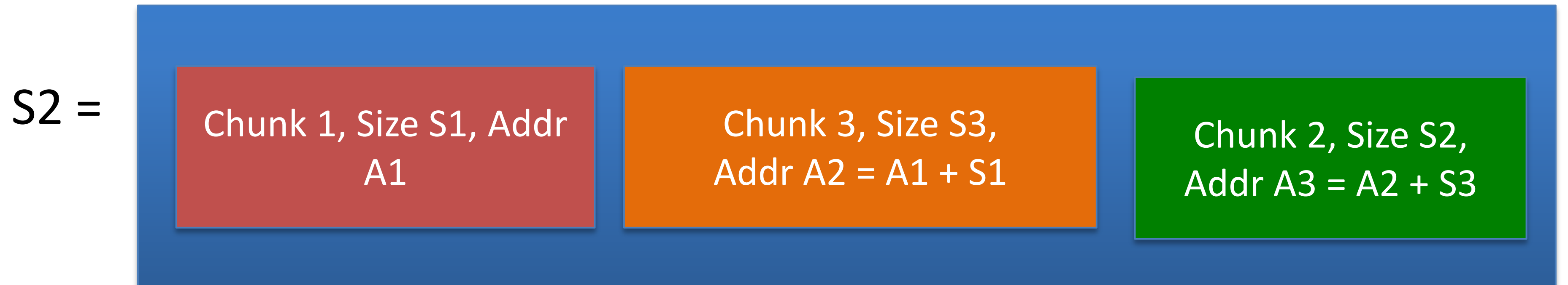
Heap in 90% of executions of program P :

S1 =



Heap in 10% of executions of program P :

S2 =

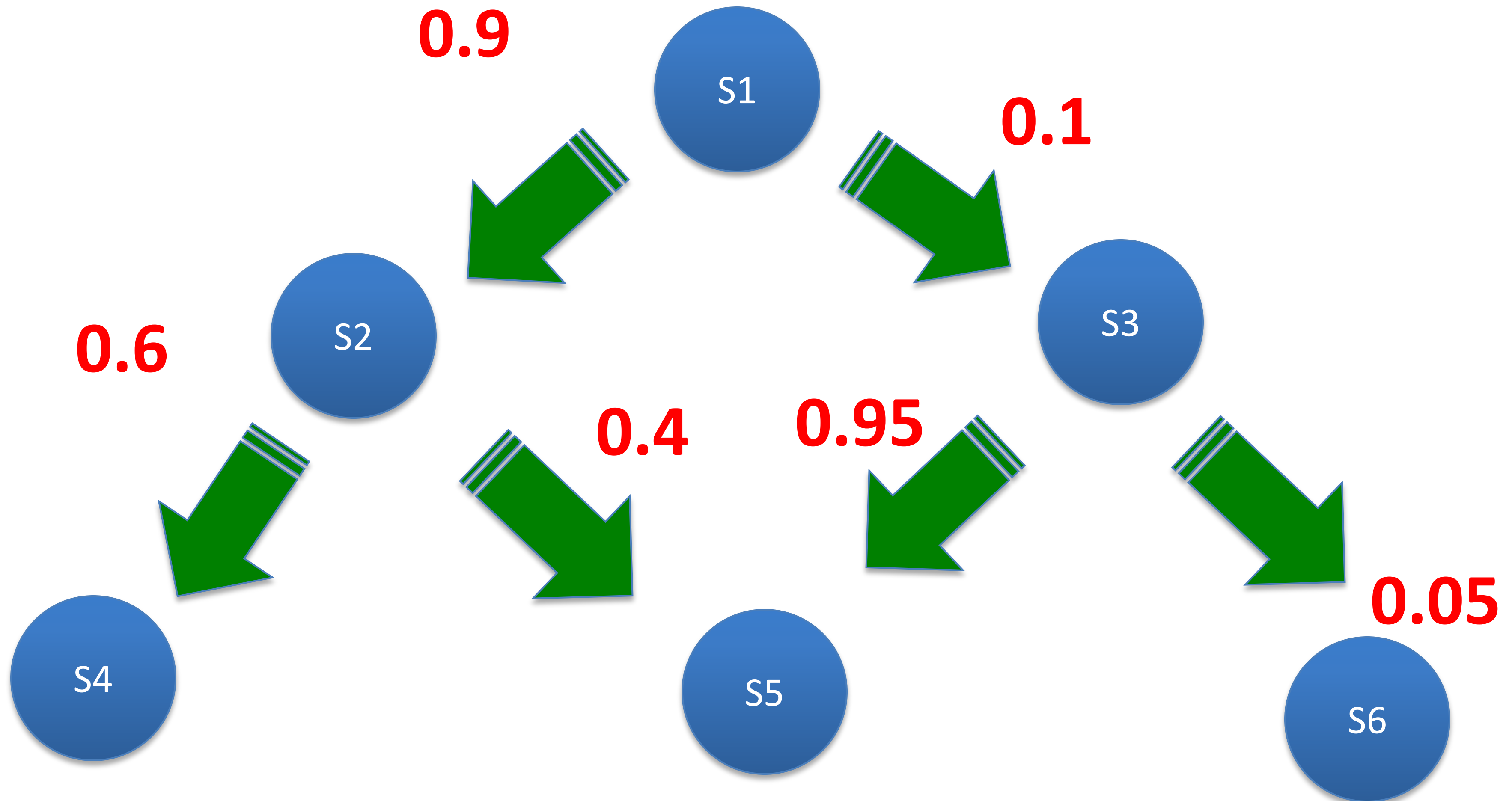


SMT solvers are unable to reason about non-determinism

Idea: Markov exploits

- Andrei Markov (1856-1922)
- Systems (Programs) may seem to act randomly, but have a hidden probabilistic regularity.
- Instrument program and deduce from sampling which paths have most chance to bring the heap in a desired exploitable state.

Markov transition system



The transition system models the set of all possible random walks.

Markov transition system

Previous slide explained:

- We computed the probability of reaching every heap states in a maximum of two heap interactions (malloc, free, etc)
- Probability of reaching S4 is:
  $P(S4) = P(S4|S2) * P(S2|S1) = 0.6 * 0.9 = 0.54$ (54%)
- Probability of reaching S5 is:
  $P(S5) = P(S5|S2) * P(S2|S1) + P(S5|S3) * P(S3|S1)$
  $= 0.9*0.4 + 0.95*0.1 = 0.455$ (45.5%)
- Probability of reaching S6 is:
  $P(S6) = P(S6|S3) * P(S3 | S1) = 0.1 * 0.05 = 0.005$ (0.5%)

Assuming S5 and S6 are the only two desired exploitable states, we can compute that the most exploitable walks end in S5.

Part 4: Synthesis

The Rise of Weird Machines

Weird Machines

- Exploitation is unintended, unexpected **computation** within the target. If the target machine were exactly as its programmers’ **model**, exploitation would not be possible.
- Exploit is a **proof by construction** that there exists a **"weird machine"**, an unexpected programmable environment inside the target. The exploit programs it.
- Features, bugs, and primitives in the target are **weird machine**’s assembly instructions.
- Weird machines don't respect layers of abstraction. **Layers of abstraction become boundaries of competence** & enable weird machines.

Any input is a program

- Sufficiently complex inputs are indistinguishable from **byte code** of a (weird) virtual machine embedded in input handler
- Input handler for sufficiently complex inputs are indistinguishable from a (weird) VM
- **Inputs run on code:**
  - All **exploits** are example of such inputs
  - ELF relocation tables are Turing Complete (TC)
  - IA32 MMU tables are TC

Exploits are models

- What do we call forgetting much of the program and focusing on just some parts?
  - (a) modeling
  - (b) symbolic execution
  - (c) exploitation?
- For example, **Return Oriented Programming** ignores (almost) all parts of the target **except** those behind its control flow graph.

Is Assembly Programming Art?

- Yes, a dark art ☺
- But, art + **algorithms** = science, right?
- **Compilation** to assembly is surely science
  - "I'd rather write programs that write programs than write programs"
- So is **automating** "weird assembly" programming
  - "When the going gets weird, the weird turn pro"

Exploits: “physics” of security

- What do we call exploration of reality beyond current mathematical abstractions?
  - (a) In Natural Science: **Physics**
  - (b) In Computer Security: **Exploitation**
- In time, new refinements emerge to fill the gap between abstraction and reality
  - This is how physics progressed

Security science is incomplete without the study of exploits

This panel was held for the Cyber Security Awareness Week 10th anniversary at the New York University Polytechnic School of Engineering on Metro Tech Center Campus.

https://csaw.isis.poly.edu/