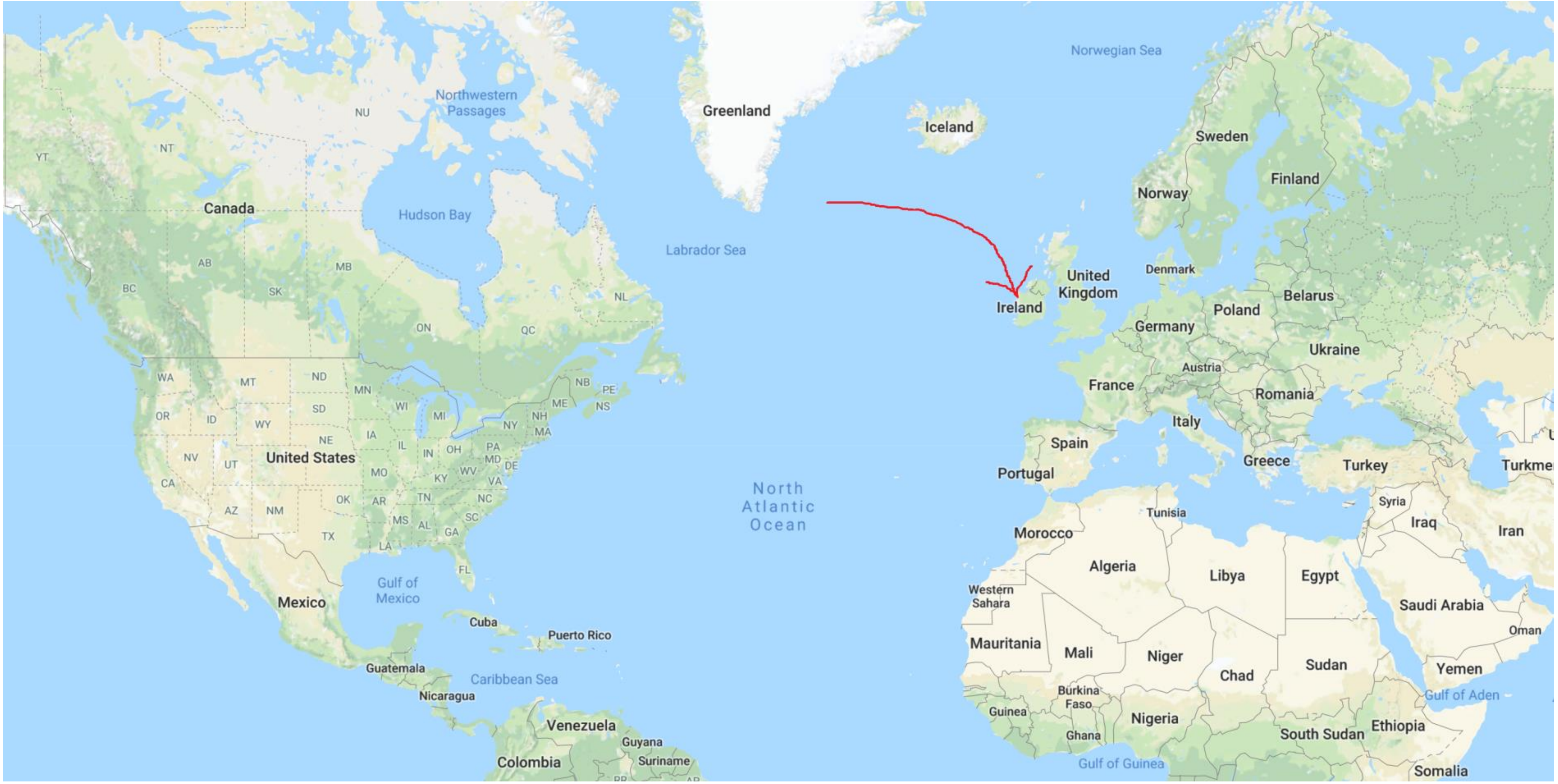
Automatic Heap Layout Manipulation

Sean Heelan

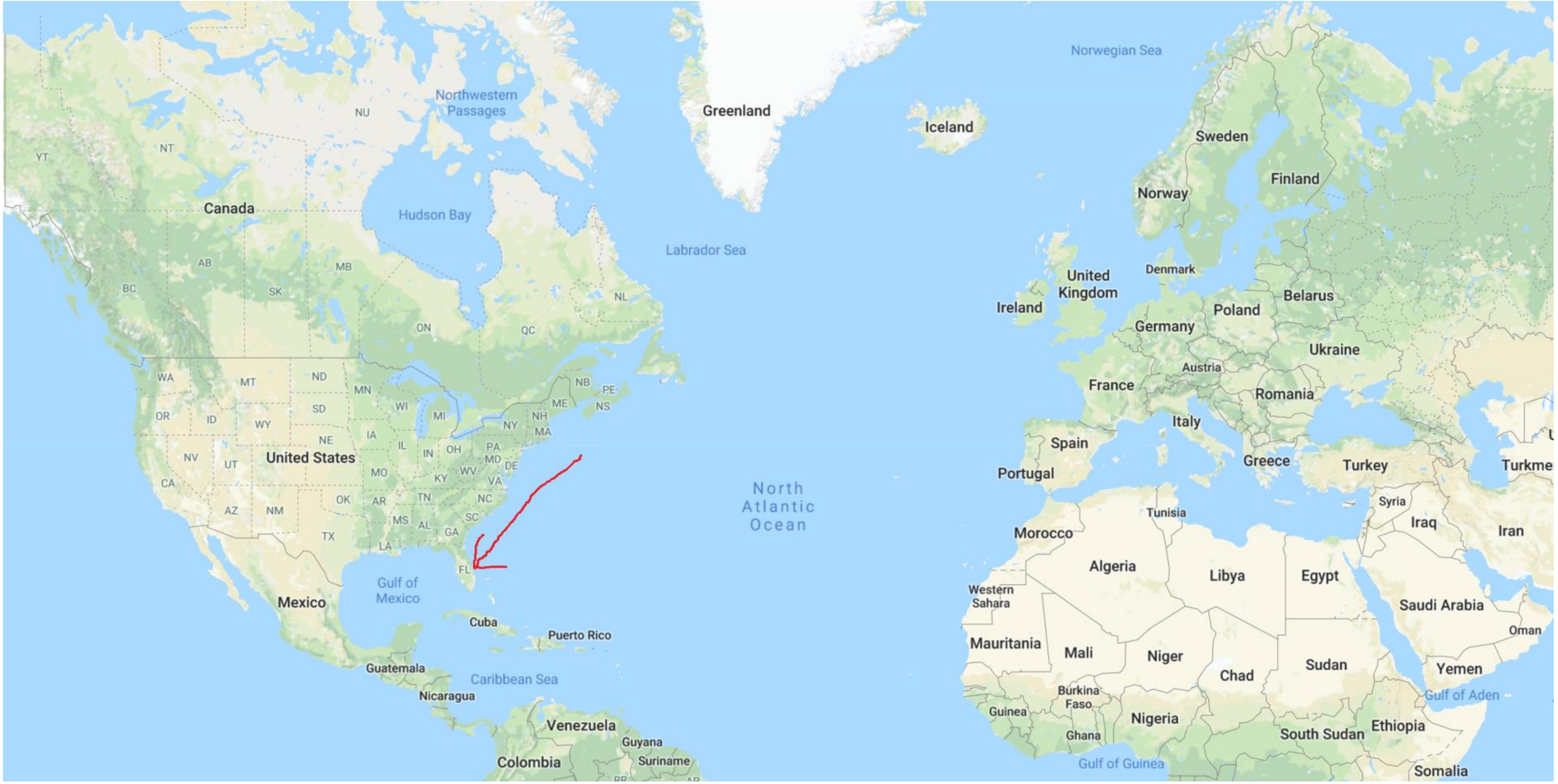
University of Oxford

https://sean.heelan.io /@seanhn / sean@vertex.re

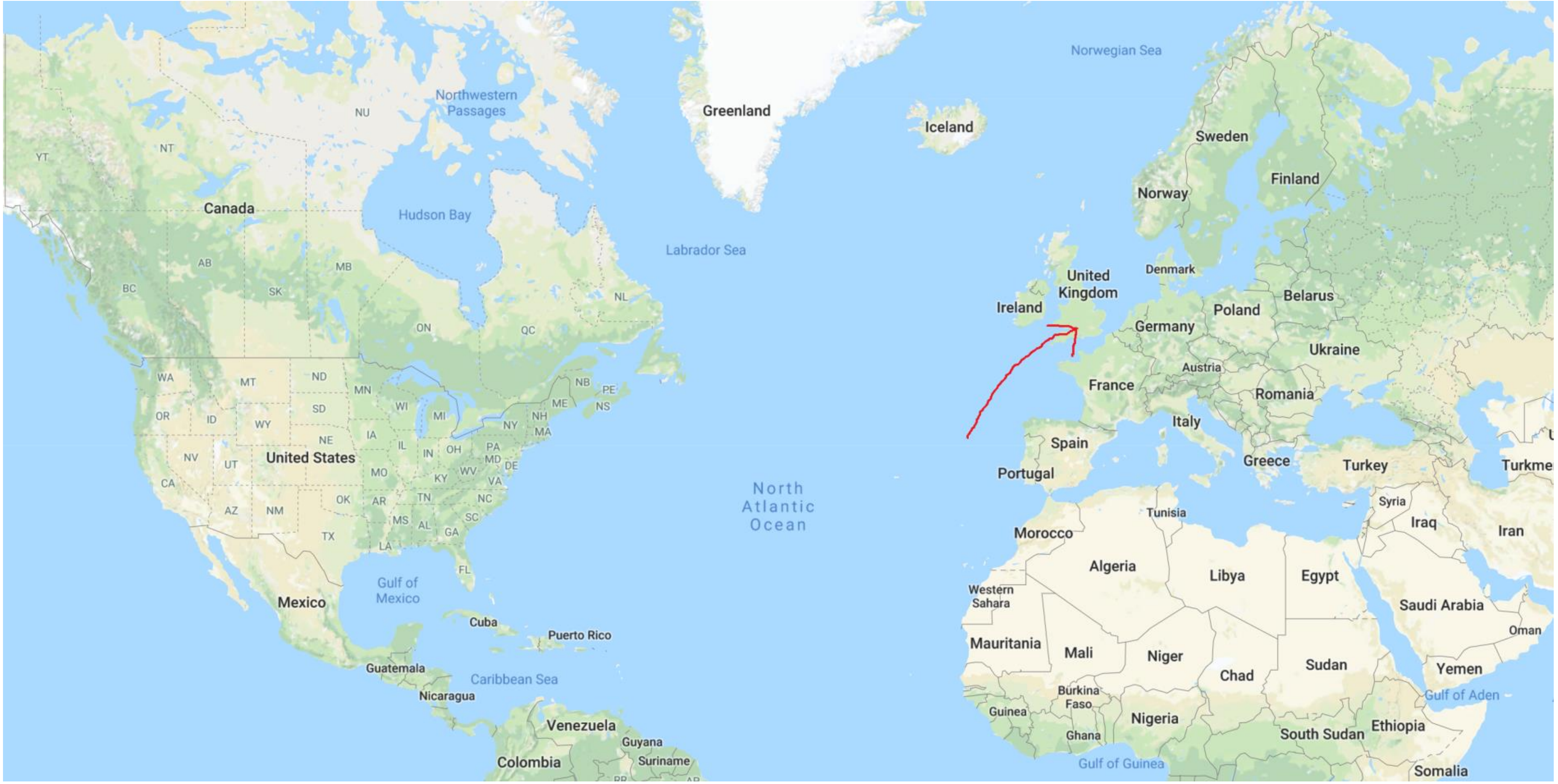
About Me


Norwegian Sea
Northwestern Passages
NU
Greenland
Iceland
Sweden
Finland
Norway
NT
YT
Canada
Hudson Bay
Labrador Sea
AB
MB
BC
SK
NL
ON
QC
United Kingdom
Ireland
Denmark
Belarus
Poland
Germany
Ukraine
Austria
France
Romania
Italy
Spain
Portugal
Greece
Turkey
Turkme
Syria
Iraq
Iran
Tunisia
Morocco
Algeria
Libya
Egypt
Western Sahara
Saudi Arabia
Oman
Mauritania
Mali
Niger
Chad
Sudan
Yemen
Gulf of Aden
Burkina Faso
Guinea
Nigeria
Ghana
South Sudan
Ethiopia
Somalia
Gulf of Guinea
North Atlantic Ocean
WA
MT
ND
MN
NB
PE
ME
NS
SD
WI
OR
ID
WY
MI
NY
NH
MA
IA
NE
United States
IL
IN
OH
PA
NV
UT
MO
MD
DE
WV
KY
VA
CA
OK
AR
TN
NC
AZ
NM
MS
AL
SC
GA
TX
LA
FL
Gulf of Mexico
Mexico
Cuba
Puerto Rico
Guatemala
Caribbean Sea
Nicaragua
Venezuela
Guyana
Suriname
Colombia

About Me


Northwestern Passages
NU
Greenland
Iceland
Norwegian Sea
Sweden
Finland
Norway
NT
YT
Canada
Hudson Bay
Labrador Sea
AB
MB
BC
SK
NL
ON
QC
United Kingdom
Denmark
Ireland
Poland
Belarus
Germany
Ukraine
Austria
France
Romania
Italy
Spain
Portugal
Greece
Turkey
Syria
Iraq
Iran
Tunisia
Morocco
Algeria
Libya
Egypt
Western Sahara
Saudi Arabia
Oman
Mauritania
Mali
Niger
Chad
Sudan
Yemen
Gulf of Aden
Burkina Faso
Guinea
Nigeria
Ghana
South Sudan
Ethiopia
Somalia
Gulf of Guinea
North Atlantic Ocean
WA
MT
ND
MN
OR
ID
SD
WI
WY
MI
NY
NH
MA
ME
NB
PE
NS
IA
NE
IL
IN
OH
PA
NV
UT
United States
MO
KY
WV
MD
DE
VA
CA
OK
AR
TN
NC
AZ
NM
MS
AL
GA
SC
TX
LA
FL
Gulf of Mexico
Mexico
Cuba
Puerto Rico
Guatemala
Caribbean Sea
Nicaragua
Venezuela
Guyana
Suriname
Colombia

About Me


Greenland
Iceland
Norwegian Sea
Northwestern Passages
NU
NT
YT
Canada
Hudson Bay
Labrador Sea
Sweden
Finland
Norway
Denmark
United Kingdom
Ireland
Poland
Belarus
Germany
Ukraine
Austria
France
Romania
Italy
Spain
Portugal
Greece
Turkey
Syria
Iraq
Iran
Tunisia
Morocco
Algeria
Libya
Egypt
Western Sahara
Saudi Arabia
Oman
Mauritania
Mali
Niger
Chad
Sudan
Yemen
Gulf of Aden
Burkina Faso
Guinea
Nigeria
Ghana
South Sudan
Ethiopia
Somalia
Gulf of Guinea
North Atlantic Ocean
United States
Mexico
Gulf of Mexico
Cuba
Puerto Rico
Guatemala
Nicaragua
Caribbean Sea
Venezuela
Guyana
Suriname
Colombia

Introduction

Background

- What is a ‘heap’?
  - An area of a program’s memory which is used to provide storage in response to dynamic memory allocation requests e.g. calls to malloc
  - Subdivided into areas of memory that are
    - ‘in use’ : Currently being used to store data by the application
    - ‘free’ : Available to service requests for memory

Background

- Physical vs Logical layout
  - Physical layout: The layout of buffers in memory, with a buffer's position given by its address
  - Logical layout: The layout of buffers in the data structures used by the allocator that determine the order in which free buffers are used to service allocation requests
    - e.g. An allocator might use a list to store the addresses of free buffers and the ordering of that list determines the order in which those buffers will be used to service allocation requests

Logical Layout Controls Physical Layout

…

Logical Layout Controls Physical Layout

…

Free32 = [0x3000, 0x2000, 0x1000]

Logical Layout Controls Physical Layout

…

Free32 = [0x3000, 0x2000, 0x1000]

vs

Free32 = [0x1000, 0x3000, 0x2000]

vs

Free32 = [0x1000, 0x2000, 0x3000]

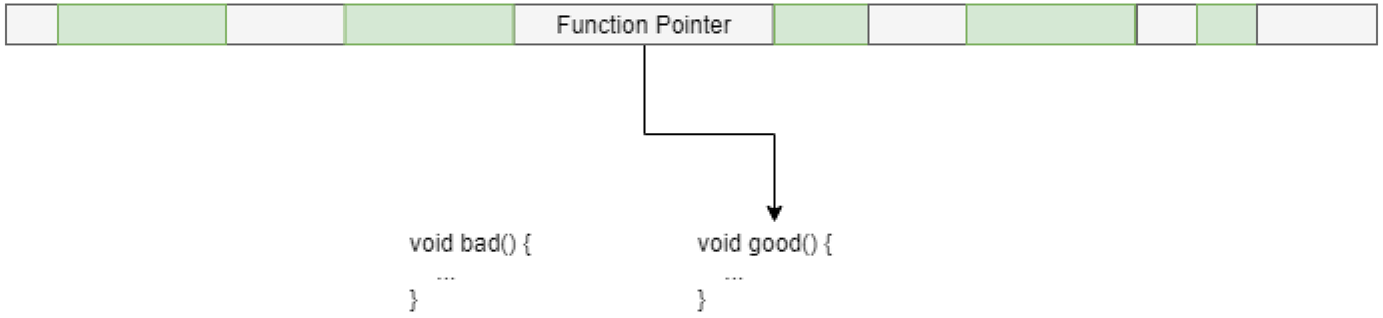
Background

- Allocators are software that manage heap space and are intended to be treated as a black-box by applications
  - i.e. Internally an allocator can use whatever data structures and algorithms it wants to manage the heap
- To predict the physical heap layout after a series of allocations and frees one needs to know the **starting state**, the series of **interactions** and the **implementation** details of the allocator

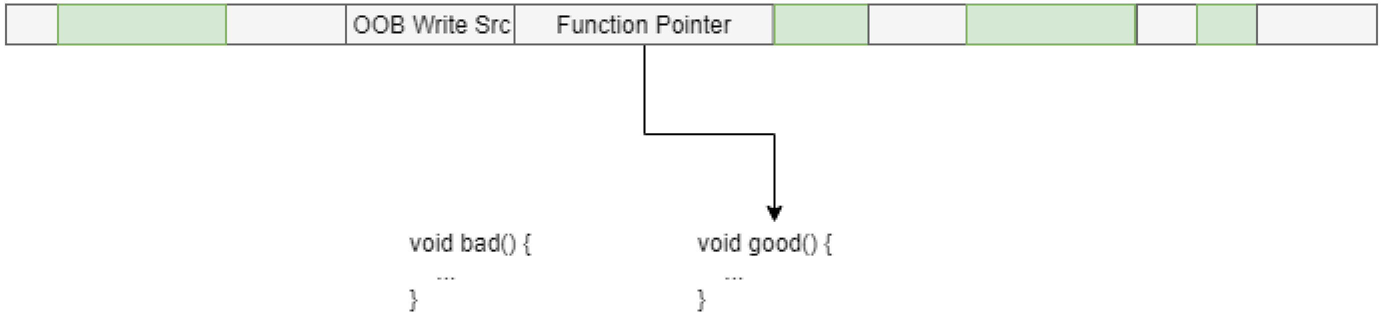
Motivation

- Assume we have
  - The ability to allocate a buffer containing a function pointer on the heap
  - The ability to trigger a heap based buffer overflow
- How do we hijack the application’s control flow?

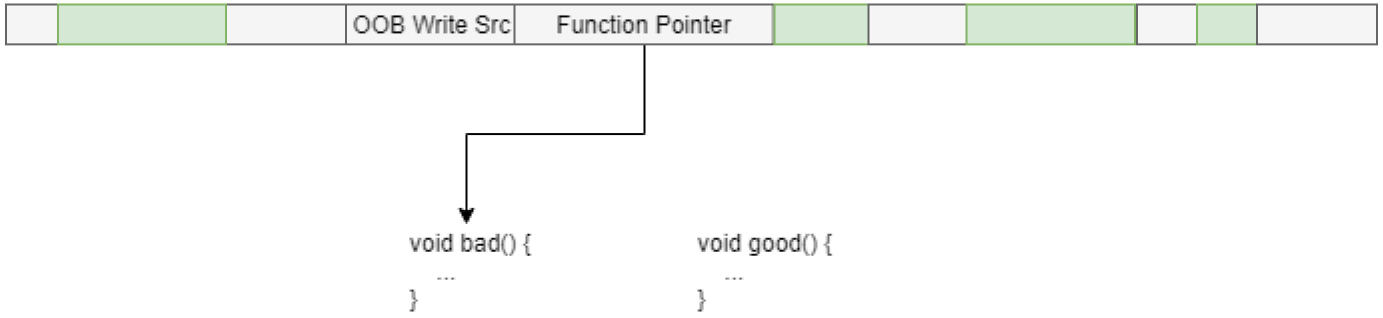
Allocate Object Containing Function Pointer



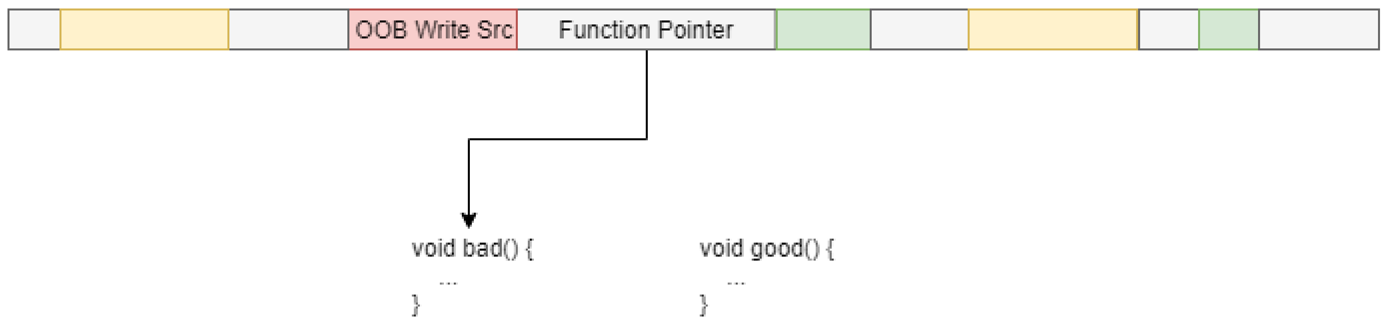
Allocate Overflow Source Buffer



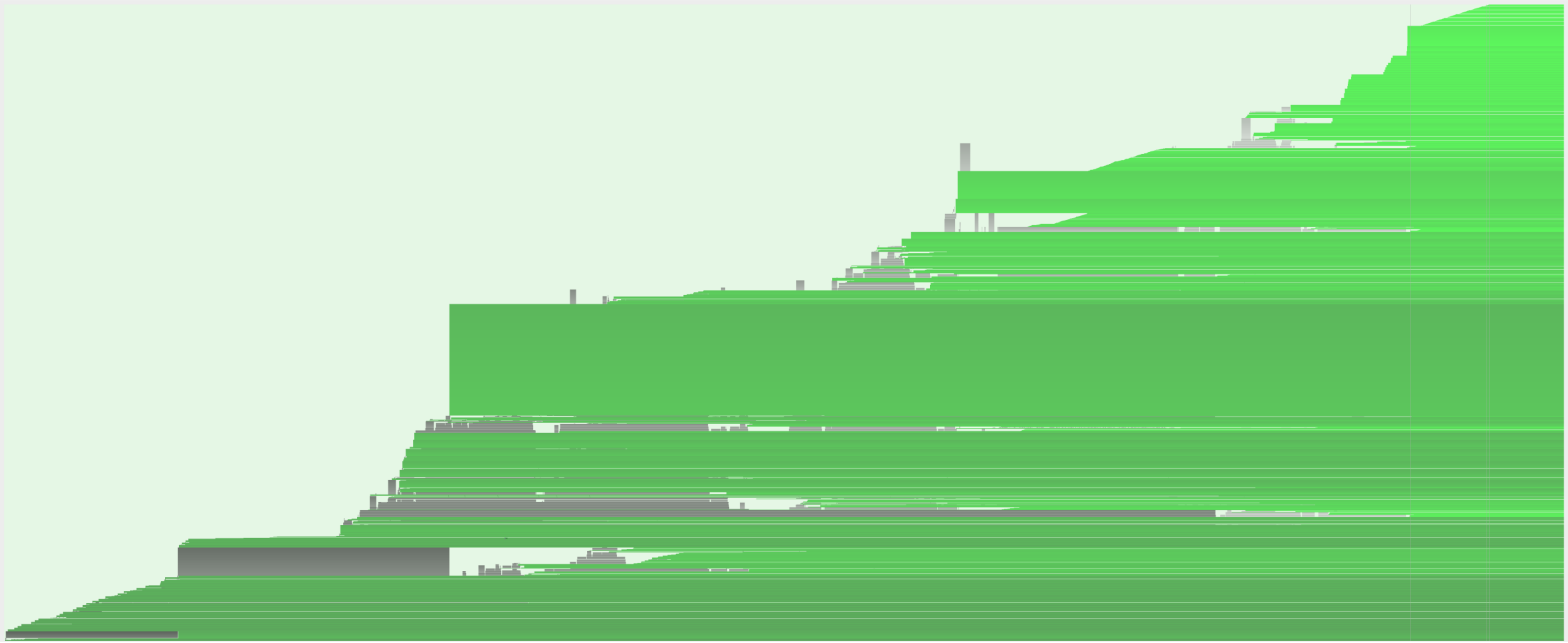
Trigger Overflow to Corrupt Pointer



But wait ...



Reality ...



Problem Overview

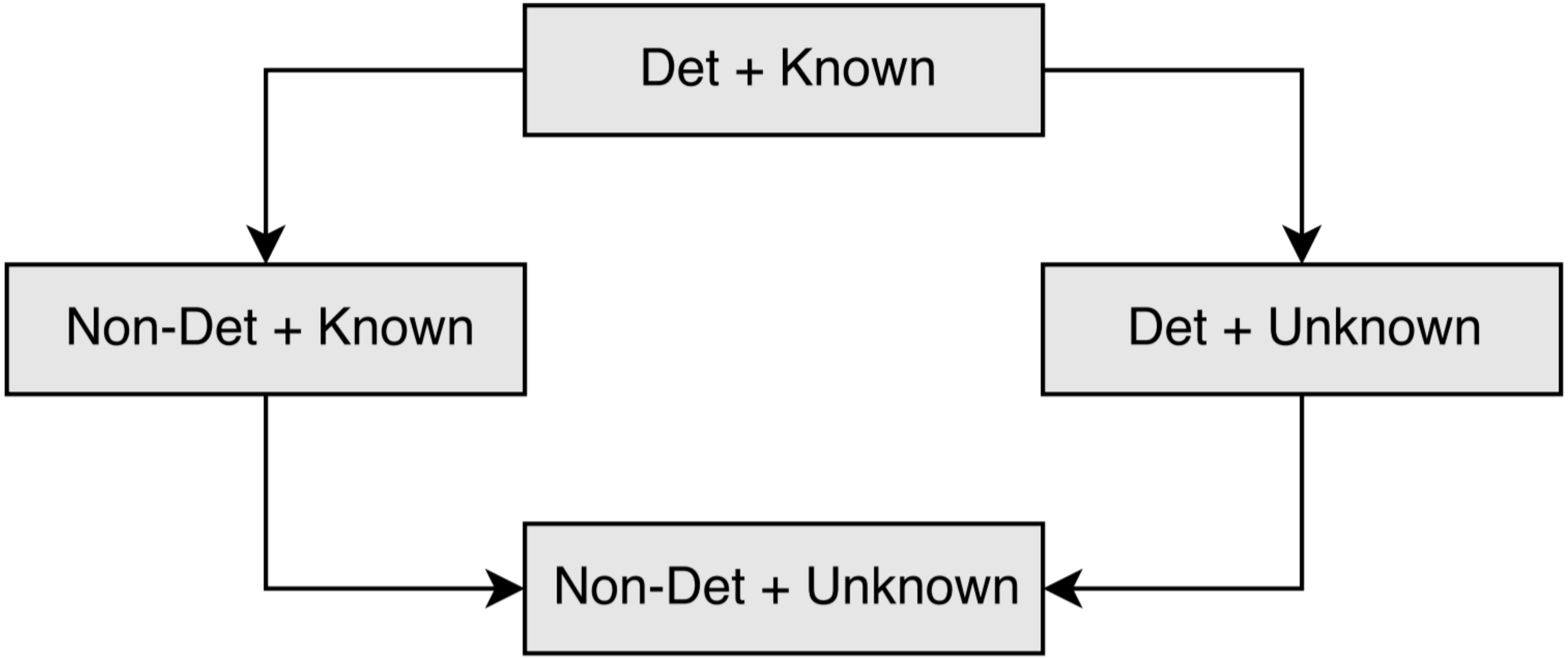
The Heap Layout Problem

- Source buffer, `S`
  - The buffer from which the overflow or underflow originates once the vulnerability is triggered
- Destination buffer, `D`
  - The buffer which we wish to corrupt once the vulnerability is triggered
- The Heap Layout Problem
  - Position `S` relative to `D` such that
    - `addressof(S) - addressof(D) = X`
    - Where X is the distance S must be from D in order for the vulnerability to corrupt the desired offset in D
    - e.g. to search for an input to position S and D immediately adjacent to each other X is set to 0

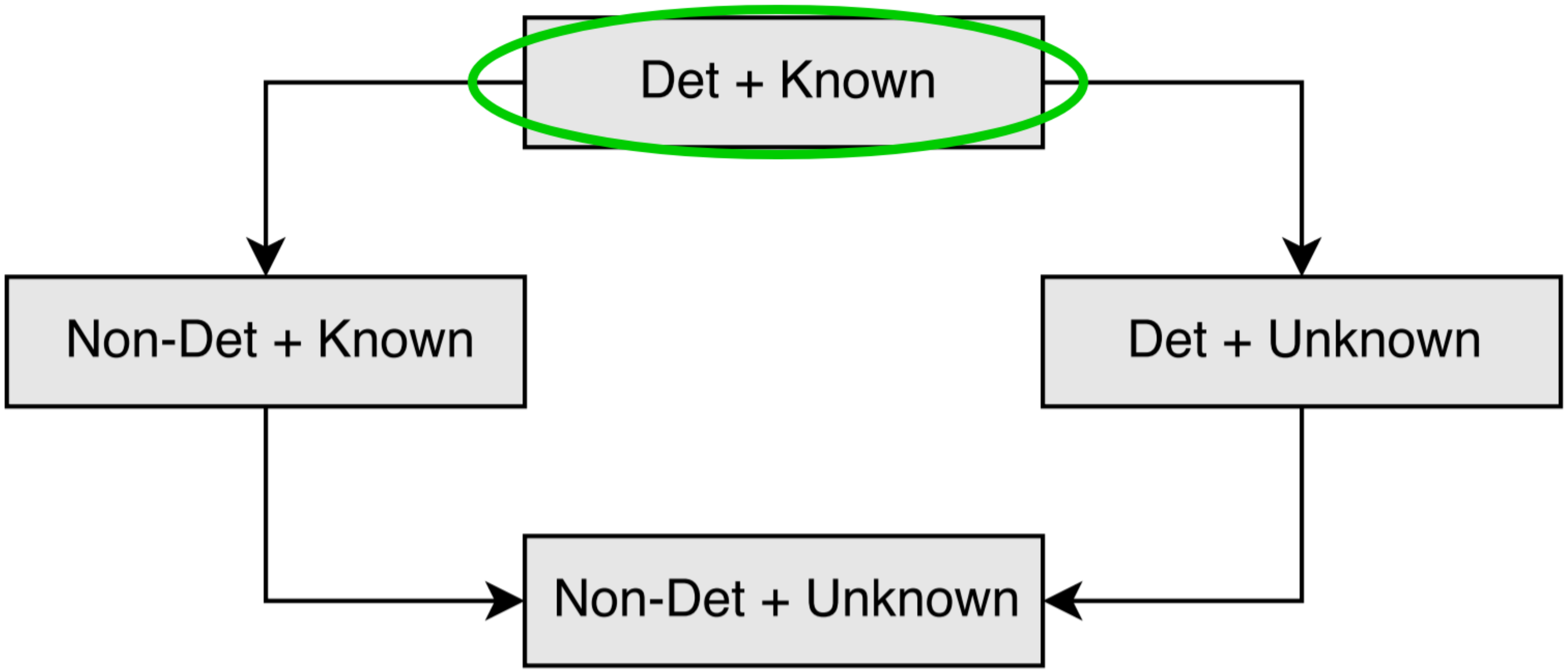
Problem Setting & Restrictions

- Deterministic allocator
  - The allocator's behaviour must be deterministic
  - Holds for a significant number of allocators, e.g. dlmalloc, tcmalloc
  - Some notable exceptions, e.g. jemalloc, Windows system allocator
- Known starting state
  - Attacker must be able to determine the starting state of the heap, or (re)set it to a known state
  - More significant restriction. Holds for many locals, and some remotes/clientsides if the attacker can trigger the creation of a new process/heap with a known initialisation sequence.
- No other actors interacting with the allocator, or the processes address space, at the same time (or if there is then their actions are deterministic)

Problem Variants



Problem Variants



Challenges to Automatic Solutions

- Allocators do not provide an API to specify relative positioning

Challenges to Automatic Solutions

- Allocators do not provide an API to specify relative positioning
- Allocators are designed to optimise different measures of success and thus utilise a diverse array of data structures and algorithms

Challenges to Automatic Solutions

- Allocators do not provide an API to specify relative positioning
- Allocators are designed to optimise different measures of success and thus utilise a diverse array of data structures and algorithms
- Applications do not typically expose a direct interface with the allocator they use

Challenges to Automatic Solutions

- Allocators do not provide an API to specify relative positioning
- Allocators are designed to optimise different measures of success and thus utilise a diverse array of data structures and algorithms
- Applications do not typically expose a direct interface with the allocator they use
- Interaction sequences which can be triggered via the application’s API are often limited in various ways and ‘noisy’

Challenges to Automatic Solutions

- Allocators do not provide an API to specify relative positioning
- Allocators are designed to optimise different measures of success and thus utilise a diverse array of data structures and algorithms
- Applications do not typically expose a direct interface with the allocator they use
- Interaction sequences which can be triggered via the application's API are often limited in various ways and 'noisy'
- The search space across all interaction sequences is usually astronomically large

SIEVE

An Evaluation Framework for Solutions to the Heap Layout Problem

Automatic Heap Layout Manipulation

- On real targets, automatic heap layout manipulation involves solutions to a number of distinct problems
    1. Figure out how to interact with the allocator via the program's API
    2. Figure out how to allocate interesting corruption targets on the heap
    3. Figure out how to solve the heap layout problem
- We can address all three of these problems separately
- SIEVE is a framework for constructing synthetic benchmarks for the heap layout problem, and evaluating solutions

The Heap Layout Problem

- Unknown complexity class*
  - Has aspects which are similar to a number of problems that are known to be NP-hard
  - e.g. the coin problem, subset sum problem, knapsack problem
- If it is NP-hard then no efficient algorithm
  - But, plenty of such problems where good enough algorithms can be built for real world application (e.g. SAT)
- SIEVE allows us to investigate the problem, and solutions, while ignoring the extra engineering involved in addressing real targets

* Apologies for the hand-waving, it’s on my To-Do list

SIEVE

- Two components
  - SIEVE driver
    - A program which links with any allocator exposing the standard malloc/free/calloc/realloc interface
    - Takes as input a series of directives
      - <malloc size ID>, <free ID>, <fst size>, <snd size> …
    - Translates the directives into function calls on the allocator
    - Outputs `addressof(fst) - addressof(snd)`
  - SIEVE framework
    - Python API for managing different experimental configurations, implementing a search algorithm, launching the driver and managing interaction with it

Creating Benchmarks in SIEVE

- A heap layout problem is parameterised by the following
  - The allocator
  - The starting state of the heap
  - The available interaction sequences with the allocator which can be triggered
  - The interaction sequence to allocate the source buffer
  - The interaction sequence to allocate the destination buffer
  - The temporal order in which the source and destination must be allocated
- SIEVE provides mechanisms for controlling each of these aspects when creating a benchmark

Example Benchmark

- Allocator=dlmalloc2.8.6
- StartingState=PythonInit
- AllocSequences=[[malloc_16]], [malloc_32, malloc_32]]
- FreeSequences=[[free_0_0], [free_1_1, free_1_0]]
- FstSequence=[malloc_16]
- SndSequence=[malloc_16]
- Order=[Fst, Snd]

Example Benchmark

- Allocator=**dlmalloc2.8.6**
- StartingState=PythonInit
- AllocSequences=[[malloc_16]], [malloc_32, malloc_32]]
- FreeSequences=[[free_0_0], [free_1_1, free_1_0]]
- FstSequence=[malloc_16]
- SndSequence=[malloc_16]
- Order=[Fst, Snd]

Example Benchmark

- Allocator=dlmalloc2.8.6
- StartingState=**PythonInit**
- AllocSequences=[[malloc_16]], [malloc_32, malloc_32]]
- FreeSequences=[[free_0_0], [free_1_1, free_1_0]]
- FstSequence=[malloc_16]
- SndSequence=[malloc_16]
- Order=[Fst, Snd]

Example Benchmark

- Allocator=dlmalloc2.8.6
- StartingState=PythonInit
- **AllocSequences=[[malloc_16]], [malloc_32, malloc_32]]**
- **FreeSequences=[[free_0_0], [free_1_1, free_1_0]]**
- FstSequence=[malloc_16]
- SndSequence=[malloc_16]
- Order=[Fst, Snd]

Example Benchmark

- Allocator=dlmalloc2.8.6
- StartingState=PythonInit
- AllocSequences=[[malloc_16]], [malloc_32, malloc_32]]
- FreeSequences=[[free_0_0], [free_1_1, free_1_0]]
- **FstSequence=[malloc_16]**
- **SndSequence=[malloc_16]**
- Order=[Fst, Snd]

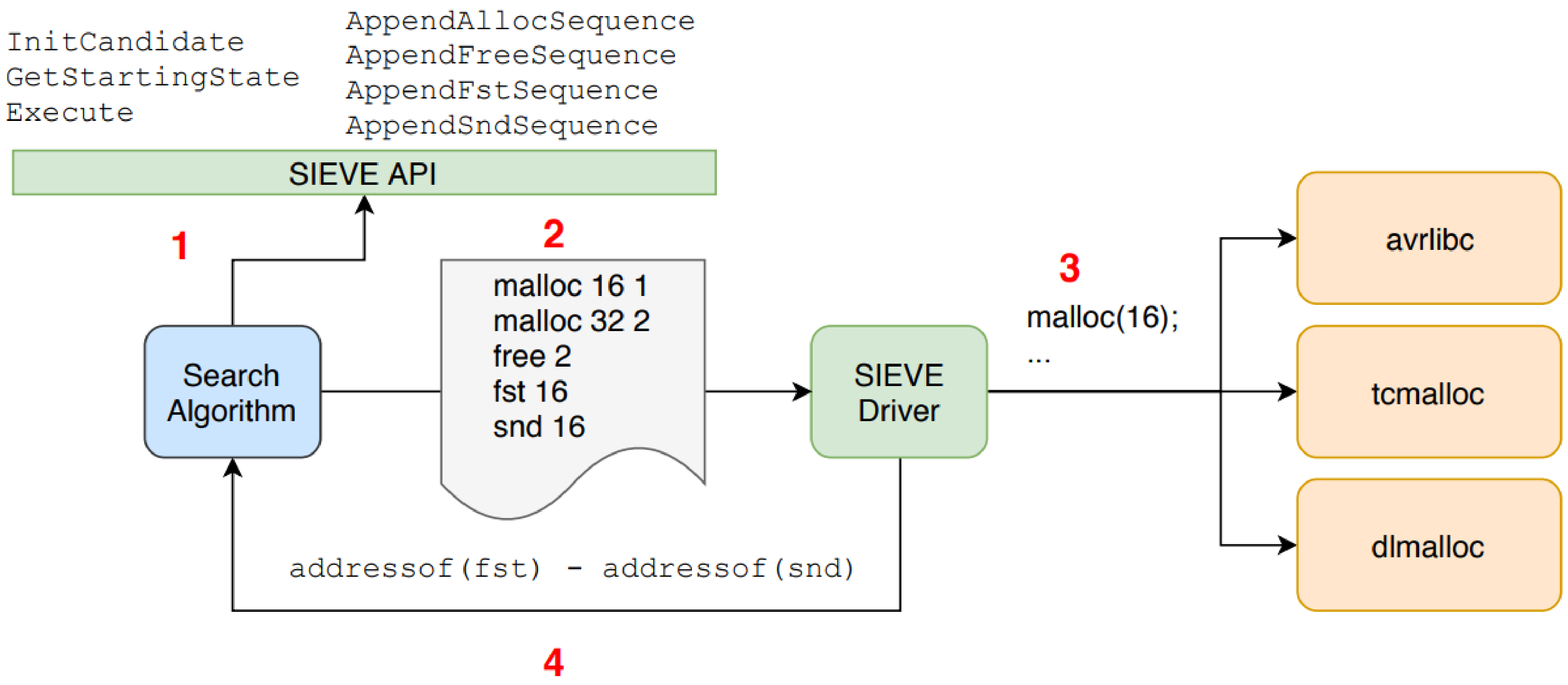
Example Benchmark

- Allocator=dlmalloc2.8.6
- StartingState=PythonInit
- AllocSequences=[[malloc_16]], [malloc_32, malloc_32]]
- FreeSequences=[[free_0_0], [free_1_1, free_1_0]]
- FstSequence=[malloc_16]
- SndSequence=[malloc_16]
- **Order=[Fst, Snd]**

Evaluating Algorithms with SIEVE

- A search algorithm in SIEVE is responsible for constructing candidate solutions,
  - e.g. a sequence of allocation and free requests to be passed to the driver
- Search algorithm can be agnostic to the benchmark configuration
  - Implemented using the SIEVE API such that it can run on arbitrary allocators and starting states, and with whatever interaction sequences are made available
- SIEVE then provides a harness for executing the defined search algorithm on a series of benchmarks

SIEVE



InitCandidate
GetStartingState
Execute
AppendAllocSequence
AppendFreeSequence
AppendFstSequence
AppendSndSequence
SIEVE API
1
2
malloc 16 1
malloc 32 2
free 2
fst 16
snd 16
Search Algorithm
SIEVE Driver
3
malloc(16);
...
avrlibc
tcmalloc
dlmalloc
addressof(fst) - addressof(snd)
4

Algorithms for the Heap Layout Problem

Random Search

- Despite astronomical search space, the solution space has a lot of symmetry
  - We only care about relative positioning for the source and destination, not absolute positioning
  - We don’t care about the positioning of holes at all, only their existence
- We can guide the search
  - e.g. by having a higher probability of selecting interaction sequences containing allocs than frees (as filing chunks is often quite useful)
- Random search requires little effort to implement
  - Even if it only works sometimes they payoff would be worthwhile

Random Search in SIEVE

```
function SEARCH(g, d, m, r)
    for i ← 0, g − 1 do
        cand ← ConstructCandidate(m, r)
        dist ← Execute(cand)
        if dist = d then
            return cand
        end if
    end for
    return None
end function
```

```
function CONSTRUCTCANDIDATE(m, r)
    cand ← InitCandidate(GetStartingState())
    len ← Random(1, m)
    fstIdx ← Random(0, len − 1)
    for i ← 0, len − 1 do
        if i = fstIdx then
            AppendFstSequence(cand)
        else if Random(1, 100) ≤ r then
            AppendAllocSequence(cand)
        else
            AppendFreeSequence(cand)
        end if
    end for
    AppendSndSequence(cand)
    return cand
end function
```

Evaluation – Benchmark Configuration

- Allocators
  - tcmalloc (v2.6.1), dlmalloc (v2.8.6), avrlibc (v2.0)
- Starting states
  - Captured the allocator interactions generated during the startup of Python and Ruby, as well as interactions between PHP and the two allocators it uses
- Source and destination sizes
  - The cross product of 8, 64, 512, 4096, 16384, 65536
- Source/Destination order
  - For each pair of sizes (x, y) run an experiment where x must be allocated temporally first and an experiment where y must be allocated temporally first

Evaluation – Benchmark Configuration

- Noise
  - Often no way to trigger a single allocator interaction at a time
  - E.g. to allocate something of size 8 maybe we have to trigger the sequence [malloc(8); malloc(16); malloc(16)]
  - The second two allocations are ‘noise’ and may make the problem more difficult to solve
- We experiment with 0, 1 and 4 noisy allocations appended onto the sequences which allocate the source and destination

Evaluation – Benchmark Configuration

- So, in total we have 2592 (3 * 4 * 36 * 2 * 3) benchmarks
  - 3 allocators, 4 starting states, 36 size pairs, 2 temporal orders, 3 noise variants
- Maximum candidates per benchmark set to 500,000
  - Translates to a maximum time per benchmark of about 15 minutes
  - This is quite short but, as we have 2592 benchmarks, it is the max feasible value given our computational resources (still takes 3 days to run everything on 40 cores =/)
  - If used 'for real', when one only needs to solve a single problem, it is likely the following results would be even better as more time can be given to the problem

Evaluation - Random Search

Allocator	Noise	% Overall Solved	% Natural Solved	% Reversed Solved
avrlibc-r2537	0	100	100	99
dlmalloc-2.8.6	0	99	100	98
tcmalloc-2.6.1	0	72	75	69
avrlibc-r2537	1	51	50	52
dlmalloc-2.8.6	1	46	60	31
tcmalloc-2.6.1	1	52	58	47
avrlibc-r2537	4	41	44	38
dlmalloc-2.8.6	4	33	49	17
tcmalloc-2.6.1	4	37	51	24

- Table presents a summary of experiments across all source/destination size combinations
- Natural
  - Given a size pair (x, y), with the constraint that x must be allocated temporally **before** y, place x physical **before** y in memory
- Reversed
  - Given a size pair (x, y), with the constraint that x must be allocated temporally **before** y, place x physical **after** y in memory

Evaluation - Random Search

Allocator	Noise	% Overall Solved	% Natural Solved	% Reversed Solved
avrlibc-r2537	0	100	100	99
dlmalloc-2.8.6	0	99	100	98
tcmalloc-2.6.1	0	72	75	69
avrlibc-r2537	1	51	50	52
dlmalloc-2.8.6	1	46	60	31
tcmalloc-2.6.1	1	52	58	47
avrlibc-r2537	4	41	44	38
dlmalloc-2.8.6	4	33	49	17
tcmalloc-2.6.1	4	37	51	24

- No noise and no segregated storage means easy problems

Evaluation - Random Search

Allocator	Noise	% Overall Solved	% Natural Solved	% Reversed Solved
avrlibc-r2537	0	100	100	99
dlmalloc-2.8.6	0	99	100	98
tcmalloc-2.6.1	0	72	75	69
avrlibc-r2537	1	51	50	52
dlmalloc-2.8.6	1	46	60	31
tcmalloc-2.6.1	1	52	58	47
avrlibc-r2537	4	41	44	38
dlmalloc-2.8.6	4	33	49	17
tcmalloc-2.6.1	4	37	51	24

- No noise and no segregated storage means easy problems
- Segregated storage significantly increases problem difficulty

Evaluation - Random Search

Allocator	Noise	% Overall Solved	% Natural Solved	% Reversed Solved
avrlibc-r2537	0	100	100	99
dlmalloc-2.8.6	0	99	100	98
tcmalloc-2.6.1	0	72	75	69
avrlibc-r2537	1	51	50	52
dlmalloc-2.8.6	1	46	60	31
tcmalloc-2.6.1	1	52	58	47
avrlibc-r2537	4	41	44	38
dlmalloc-2.8.6	4	33	49	17
tcmalloc-2.6.1	4	37	51	24

- No noise and no segregated storage means easy problems
- Segregated storage significantly increases difficulty
- The addition of a single noisy allocation significantly increases difficulty

Evaluation - Random Search

Allocator	Noise	% Overall Solved	% Natural Solved	% Reversed Solved
avrlibc-r2537	0	100	100	99
dlmalloc-2.8.6	0	99	100	98
tcmalloc-2.6.1	0	72	75	69
avrlibc-r2537	1	51	50	52
dlmalloc-2.8.6	1	46	60	31
tcmalloc-2.6.1	1	52	58	47
avrlibc-r2537	4	41	44	38
dlmalloc-2.8.6	4	33	49	17
tcmalloc-2.6.1	4	37	51	24

- No noise and no segregated storage means easy problems
- Segregated storage significantly increases difficulty
- The addition of a single noisy allocation significantly increases difficulty
- The allocation order to temporal order relationship matters

Evaluation - Random Search

Allocator	Noise	% Overall Solved	% Natural Solved	% Reversed Solved
avrlibc-r2537	0	100	100	99
dlmalloc-2.8.6	0	99	100	98
tcmalloc-2.6.1	0	72	75	69
avrlibc-r2537	1	51	50	52
dlmalloc-2.8.6	1	46	60	31
tcmalloc-2.6.1	1	52	58	47
avrlibc-r2537	4	41	44	38
dlmalloc-2.8.6	4	33	49	17
tcmalloc-2.6.1	4	37	51	24

- No noise and no segregated storage means easy problems
- Segregated storage significantly increases difficulty
- The addition of a single noisy allocation significantly increases difficulty
- The allocation order to temporal order relationship matters (with a caveat for avrlibc)

Conclusion – Random Search

- Random search performs very well when there is no noise, and no segregated storage
- As noise increases, or with the addition of segregated storage, random search begins to struggle
  - Worst case, down to 17% of problems solved
- Note: The previous results were an average across 10 runs of each set of benchmarks (e.g. each of the 2592 benchmarks was run 10 times and an average taken)
  - If we consider all 10 runs, then 78% of benchmarks were solved at least once
  - i.e. only 22% of benchmarks were never solved if the execution budge was 5 million candidates instead of 500,000
  - With appropriate computational resources random search is pretty effective

A Better Algorithm for the Heap Layout Problem

- While random search has a high pay out, considering it’s simplicity, almost ¼ of the problems were never solved
- How can we improve?
  1. Generate a logical model of the allocator and apply a SAT solver
     - May work for small allocators, likely to struggle beyond that
  2. Apply MCTS as used successfully for Go, Poker etc.
     - The more impressive results involving MCTS are from large, complex, systems, and the engineering effort required to get good results on real problems is non-obvious to me
     - e.g. AlphaGo uses MCTS but also a variety of other networks and optimisations

A Genetic Algorithm for the Heap Layout Problem

- Genetic Algorithms offer a flexible solution to optimisation/search problems
- A large number of real world examples of successful application, e.g. many fuzzing implementations can be characterised in this way
- Requirements:
  - A representation for individuals in the population
  - Mutation and crossover operators
  - A fitness function
  - A selection algorithm

GA Details

- Implemented on top of the Distributed Evolutionary Algorithm Platform (DEAP) (In Python)
  - https://github.com/DEAP/deap
- Upsides
  - Comes with several genetic algorithms that can be used out of the box
  - Comes with a lot of useful auxiliary functions, such as selection algorithms, statistics gathering at runtime etc
  - Designed to operate in a distributed fashion by default, so scalable
- Downsides
  - Uses a lot of expensive deep copy operations internally
  - The core algorithms fail to exploit parallelism in places where it would be useful
    - e.g. mutation/crossover are done sequentially across the population

GA Details – Individual Representation

- Individuals represent a series of allocation and free requests (i.e. a candidate to be sent to the SIEVE driver)
  - Each allocation has a size and ID argument, while each free has an ID argument
- In practice we use a Python `array.array` of integers to represent these

GA Details – Fitness and Selection

- Instead of using a single fitness score and optimizing that, we optimize a multiobjective function over the following two objectives:
  - `minimise(distance(fst, snd))`
    - If the allocations are in the wrong order then the distance function returns `MAX_ERROR`
  - `minimise(len(individual))`
    - To prioritise shorter over longer individuals of the same fitness
- NSGA-II selection algorithm used to rank individuals based on multiple objectives

GA Details – Mutation

- Mutation: Given a single individual, apply between 1 and N of the following mutations to produce a new individual
  - Select a up to M operations in the individual and mutate them by changing allocs to frees, frees to allocs, alloc sizes and free targets
  - Move the first allocation of interest to a randomly selected new position
  - Shorten the individual by removing a random interval of operations from it
  - Select two non-overlapping intervals and swap them
  - Add a randomly generated sequence of allocations and frees at a random index
  - Add a ‘hole pattern’
  - Add a ‘spray pattern’
  - Add a ‘nudge’ – a single (or low number) of allocations or frees

Spray Patterns

- From prior experience of solving heap layout problems manually we know that there are techniques that tend to be useful in general
- Filling chunks on the heap
  - Helps normalise the heap and remove free chunks that would capture our allocations of interest
  - The GA can elect to add a ‘spray pattern’ which is a large number of allocations of the same size

Hole Patterns

- Creating a hole can be useful to capture noise, or sometimes to position allocations of interest
- Certain patterns of allocations and frees tend to create holes, as long as the allocations end up adjacent, e.g. 3 success allocations followed by a free of the middle allocation
- The GA can elect to add sequences crafted to create holes

GA Details - Crossover

- Given two individuals, select an interval from each and transplant them
- Intended to produce solutions by taking useful components from multiple partial solutions

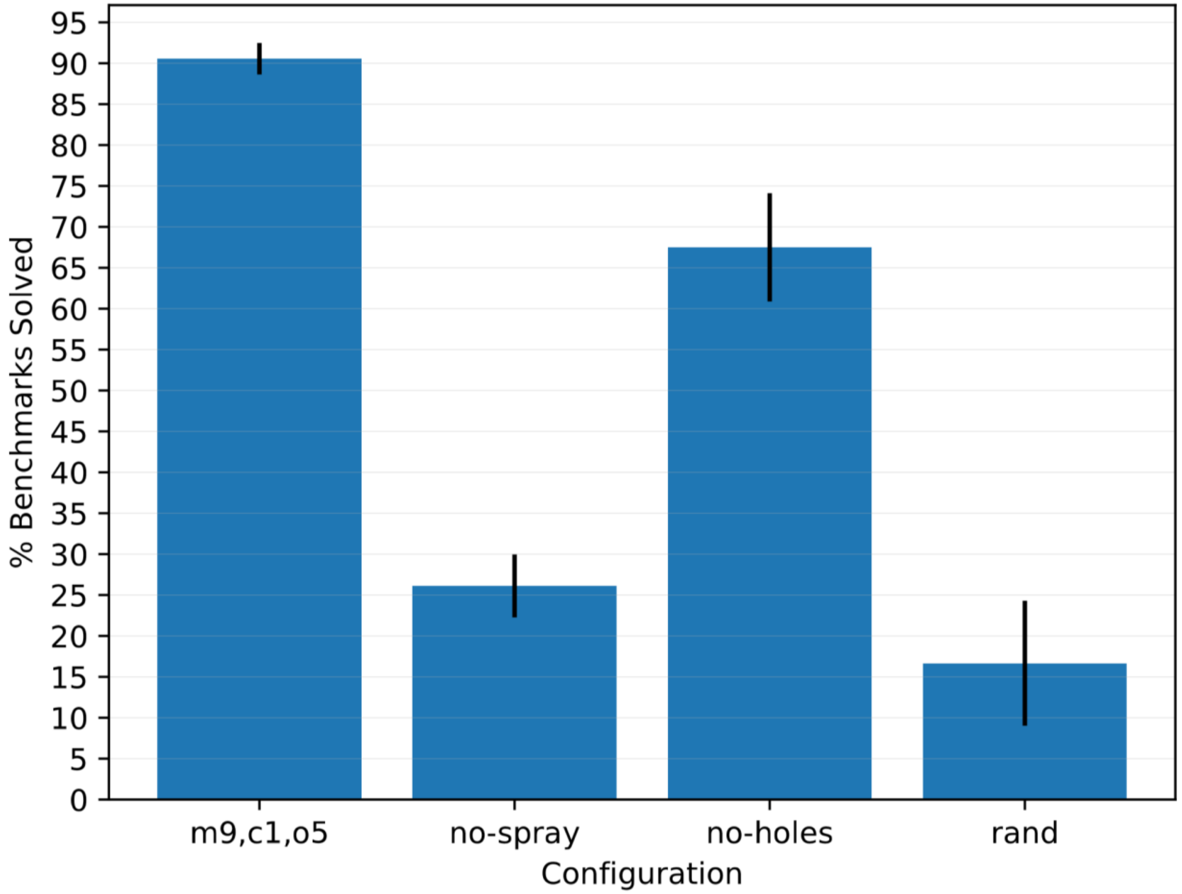
Evaluation – GA Configuration

- Population size: 200, Generation count: 500
  - Translates to a maximum of 100,000 individuals produced
- Evolutionary algorithm: $\mu + \lambda$
  - Given a population, produce the next generation by applying mutation and crossover to **produce $\boldsymbol{\lambda}$ new individuals** and then from those plus the initial population **select $\boldsymbol{\mu}$**
- Selection algorithm: NSGA-II
- The probability of applying mutation or crossover, and the individual mutation operations, is controlled via a configuration file
  - Multiple configurations tried to determine the best

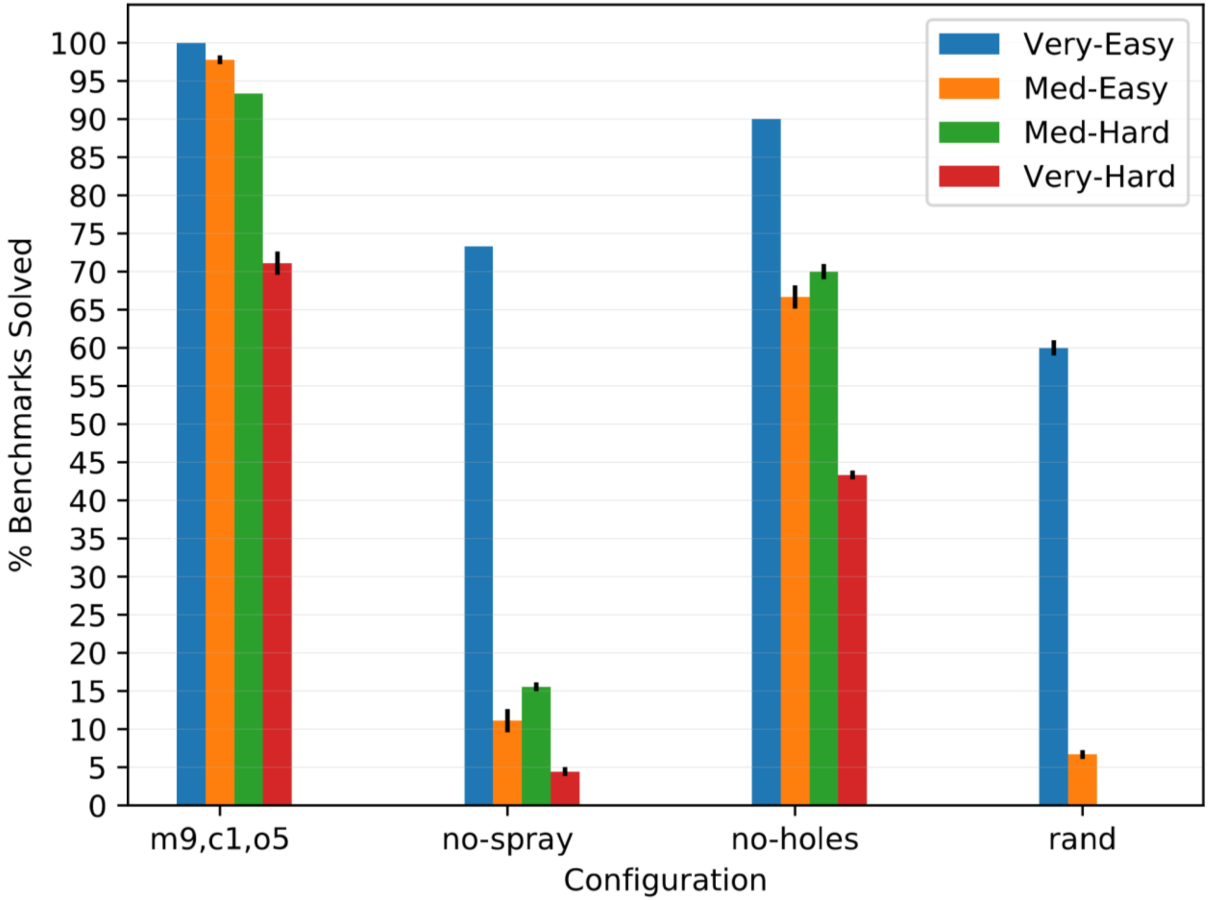
Evaluation – Selecting Benchmarks

- From the 2.5k earlier benchmarks we have categorised them based on the number of runs they were solved in by random search
  - Always Solved (Very-Easy)
  - Never Solved (Very-Hard)
  - Solved on 30%-40% of experimental repetitions (Med-Hard)
  - Solved on 60%-70% of experimental repetitions (Med-Easy)
- In each category we selected 15 benchmarks, aiming for an even distribution in each category across allocators, starting states, and sizes
- Each benchmark ran 3 times, results presented are an average

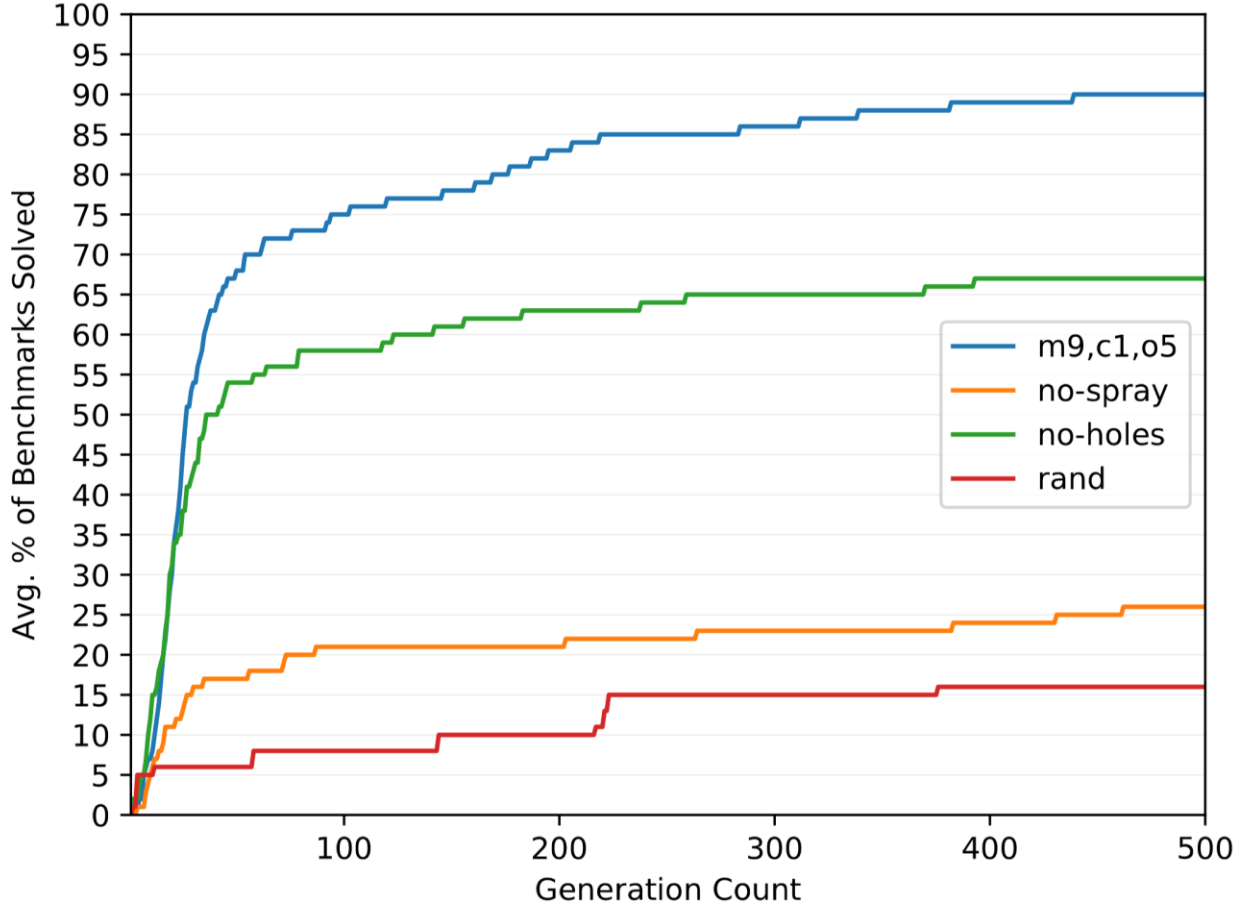
Avg. % of Benchmarks Solved



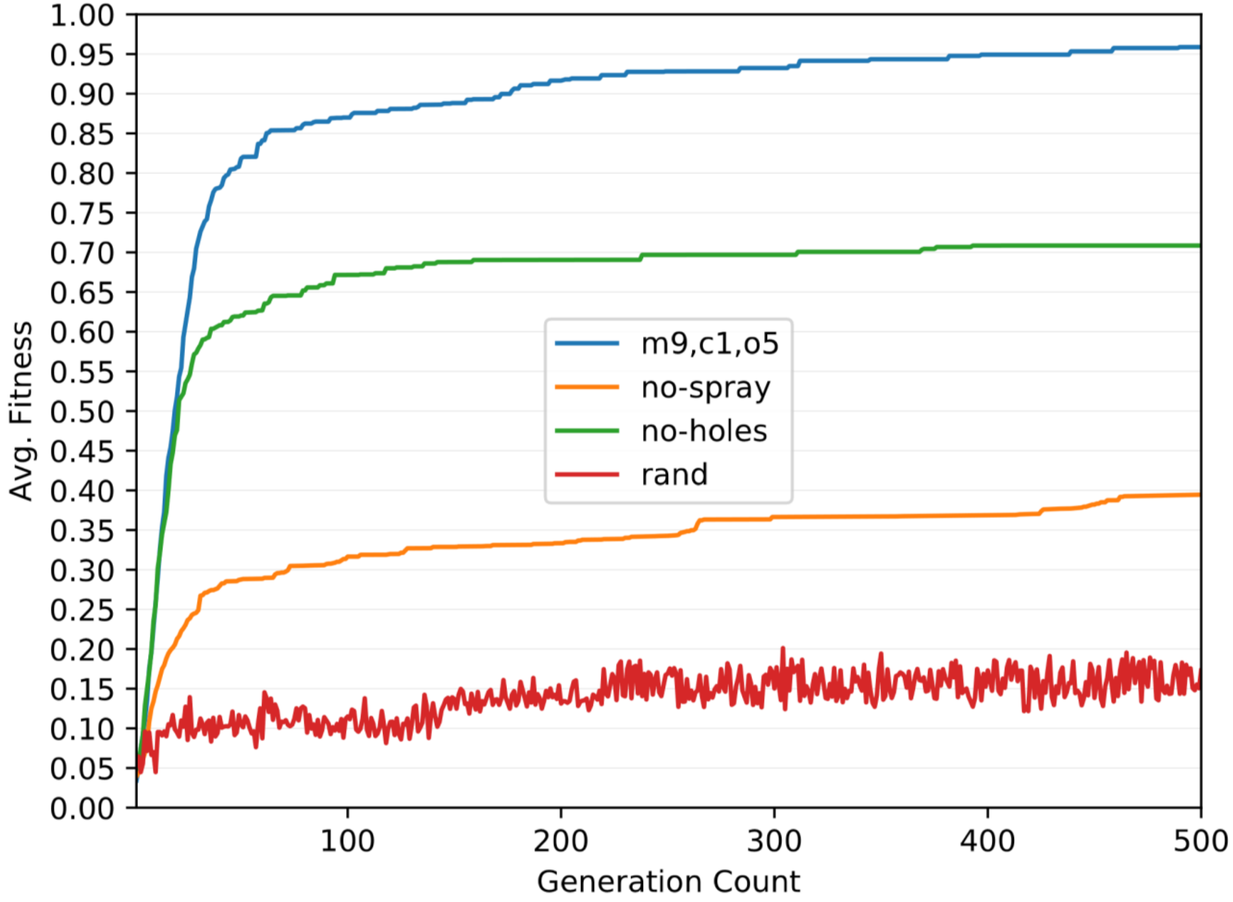
Avg. % of Benchmarks Solved per Category



Avg. % of Benchmarks Solved per Generation



Avg. Fitness (1 / distance) per Generation

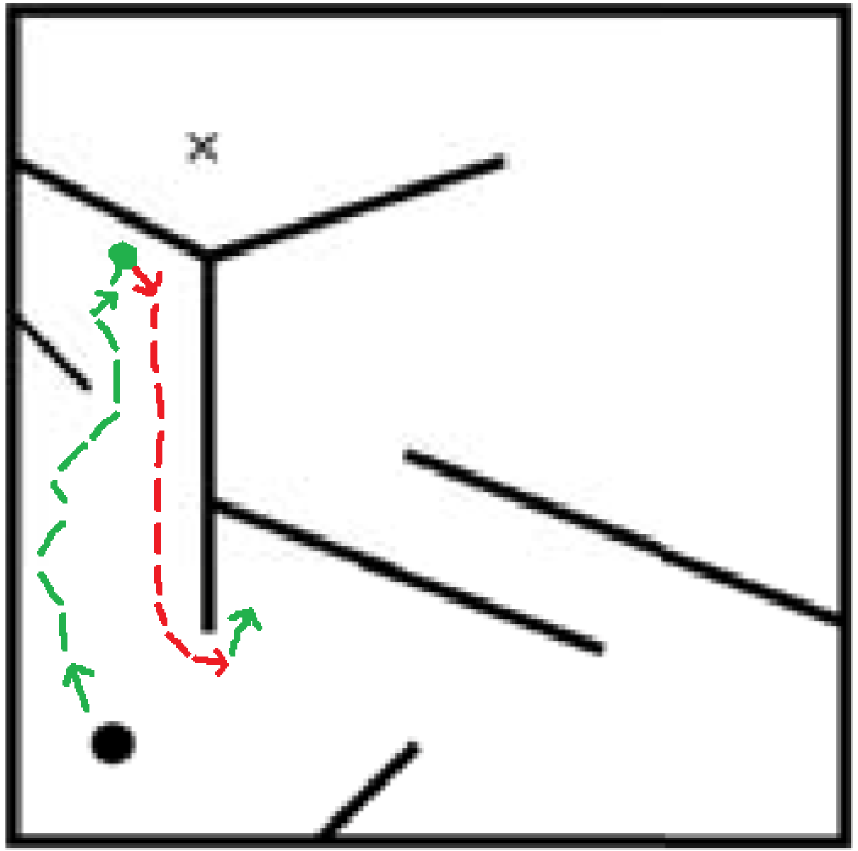


GA Conclusion

- Relatively simple to implement and significantly more powerful than random search
  - Solutions rapidly evolve with more than half the benchmarks solved in less than 50 generations
  - Can address problems that were never solved via random search, on average solving 70% of these
- Providing spray/hole pattern generation as a base mutation primitive provides a particularly strong advantage
  - Could these primitives be evolved from by making single additions of allocs or frees? Unlikely with distance as a fitness metric, but may be worth investigating.
- Of the 60 benchmarks, only 3 were never solved

Deception

- The heap layout problem, with distance as a fitness function, is what is known as a **deceptive domain**
- In some situations, the fitness function may lead us to a local optimum from which there is no way to escape towards a global optimum without a number of steps through which the fitness **decreases**
- In all cases I have investigated where the GA fails, it is due to deception



Real World Application

What do we have so far?

- Two search algorithms capable of solving instances of the heap layout problem
  - Random search – Simple to implement, fast to run
  - Genetic algorithm – Slightly more complex implementation and slower runtime, but much more powerful
- Requirements
  - Some way to interact with a target’s allocator,
  - Some way to allocate the source and destination buffers

Working on Real Programs

- Need some way to figure out how to trigger interactions with the target's allocator, then we can implement random search or the GA on top of that
- Remember, we still have our initial assumptions of a deterministic allocator and known starting state however
  - This limits the applicability of this approach but there are some viable scenarios
- For evaluation I chose the PHP language interpreter
  - Threat model: a hardened interpreter in which we can run arbitrary PHP code but want to execute native code, i.e. escape from the hardened shell

High Level Algorithm

1. Discover how to interact with the allocator via the API of the program to produce Z, a set of API calls which can be used for layout manipulation
2. Combine a series of API calls from Z and provide them to the target program, including the API call to allocate S and D
3. Check whether `addressof(S) - addressof(D) = X`, if not go to step 2, but if it is then exit and report the discovered sequence of API calls to the user

High Level Algorithm

1. **Discover how to interact with the allocator via the API of the program to produce Z, a set of API calls which can be used for layout manipulation**
2. Randomly combine a series of API calls from Z and provide them to the target program, including the API call to allocate S and D
3. Check whether `addressof(S) - addressof(D) = X`, if not go to step 2, but if it is then exit and report the discovered sequence of API calls to the user

Identifying Available Interaction Sequences

- Effectively performed by fuzzing, tuned towards discovering interaction sequences rather than bugs
  - Leverages prior work "*Ghosts of Christmas Past: Fuzzing Language Interpreters using Regression Tests*" from Infiltrate '14
- Basic idea is to deconstruct PHP's regression tests into small, valid, chunks of PHP code, then
  - Instrument the target binary to record allocs/frees that occur as a result of a particular function call
  - Utilise mutation and recombination to produce new fragments with new behaviours

Fragmentation

```
<?php
$image = imagecreatetruecolor(180, 30);
imagestring($image, 5, 10, 8, ‘Text', 0x00ff00);

$gaussian = array(
    array(1.0, 2.0, 1.0),
    array(2.0, 4.0, 2.0)
);

var_dump(imageconvolution(
        $image, $gaussian, 16, 0));
?>
```

Fragmentation

```
<?php
$image = imagecreatetruecolor(180, 30);
imagestring($image, 5, 10, 8, ‘Text', 0x00ff00);

$gaussian = array(
    array(1.0, 2.0, 1.0),
    array(2.0, 4.0, 2.0)
);

var_dump(imageconvolution(
    $image, $gaussian, 16, 0));
?>
```

```
imagecreatetruecolor(180, 30)
imagestring($image, 5, 10, 8, ‘Text’, 0x00ff00)
array(array(1.0, 2.0, 1.0), array(2.0, 4.0, 2.0))
array(1.0, 2.0, 1.0)
array(2.0, 4.0, 2.0)
var_dump(imageconvolution($image, $gaussian, 16, 0))
```

Fragmentation

```
<?php
$image = imagecreatetruecolor(180, 30);
imagestring($image, 5, 10, 8, 'Text', 0x00ff00);

$gaussian = array(
    array(1.0, 2.0, 1.0),
    array(2.0, 4.0, 2.0)
);

var_dump(imageconvolution(
    $image, $gaussian, 16, 0));
?>
```

```
imagecreatetruecolor(180, 30)
imagestring($image, 5, 10, 8, 'Text', 0x00ff00)
array(array(1.0, 2.0, 1.0), array(2.0, 4.0, 2.0))
array(1.0, 2.0, 1.0)
array(2.0, 4.0, 2.0)
var_dump(imageconvolution($image, $gaussian, 16, 0))
```

```
imagecreatetruecolor(I, I)
imagestring(R, I, I, I, T, I)
array(F, F, F)
array(R, R)
var_dump(R)
imageconvolution(R, R, I, I)
```

Synthesis

imagecreatetruecolor(I, I)
imagestring(R, I, I, I, T, I)
array(F, F, F)
array(R, R)
var_dump(R)
imageconvolution(R, R, I, I)

Synthesis

```
imagecreatetruecolor(I, I)
imagestring(R, I, I, I, T, I)
array(F, F, F)
array(R, R)
var_dump(R)
imageconvolution(R, R, I, I)
```

```
imagecreatetruecolor(1, 1)
imagecreatetruecolor(1, 2)
imagecreatetruecolor(1, 3)
imagecreatetruecolor(1, 4)
…
```

Synthesis

```
imagecreatetruecolor(I, I)
imagestring(R, I, I, I, T, I)
array(F, F, F)
array(R, R)
var_dump(R)
imageconvolution(R, R, I, I)
```

```
imagecreatetruecolor(1, 1)
imagecreatetruecolor(1, 2)
imagecreatetruecolor(1, 3)
imagecreatetruecolor(1, 4)
…
```

```
imageconvolution(array(1.0, 2.0, 1.0), imagecreatetruecolor(180, 30), 16, 0)
imageconvolution(array(2.0, 4.0, 2.0), imagecreatetruecolor(180, 30), 16, 0)
imageconvolution(imagecreatetruecolor(180, 30), imagecreatetruecolor(180, 30), 16, 0)
…
```

Synthesising PHP Fragments

- From PHP’s 12k or so tests we produce 300 standalone fragments containing a single function call
- 15 minutes or so of fuzzing (80 cores) produces over 10k fragments which trigger unique allocator interaction sequences
- Varying in length from a single allocator interaction to thousands of allocator interactions per fragment

High Level Algorithm

1. Discover how to interact with the allocator via the API of the program to produce a set of API calls which can be used for layout manipulation, Z
2. **Randomly combine a series of API calls from Z and provide them to the target program, including the API call to allocate S and D**
3. Check whether `addressof(S) - addressof(D) = X`, if not go to step 2, but if it is then exit and report the discovered sequence of API calls to the user

Random Search

- Randomly combine sequences of PHP code from the fragment database to form a valid PHP program
- Insert the code to allocate D and S at a random location in the program
- Some guidance on the search (not truly random)
  - Prioritise sequences which allocate/free sizes equal to the size of D or S
  - Prioritise sequences which trigger fewer allocator interactions
  - Once both S and D are allocated stop appending fragments

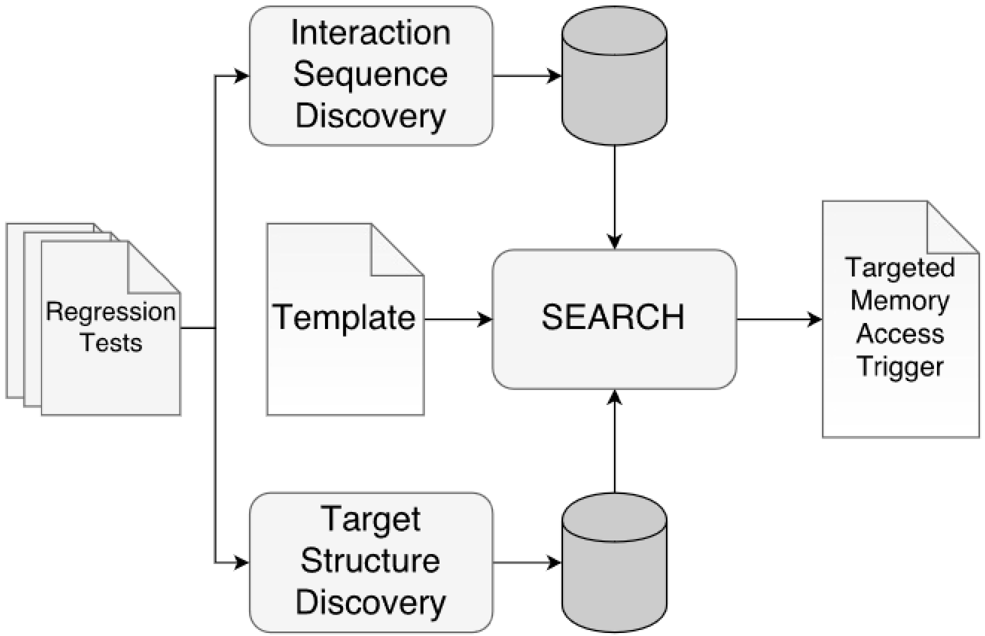
Randomly Produced Sequence

```
$var_vtx_43 = str_repeat("\f4", 106);
$var_vtx_44 = imagecreate(10, 10);
$var_vtx_45 = str_repeat("a-very-long-break-string-to-clobber-the-heap", 8);
$var_vtx_46 = unserialize('a:2:{i:0;O:12:"DateInterval":1:{s:1:"y";R:1;}i:1;i:2;}');
$var_vtx_47 = str_repeat("747 X ", 58);
$var_vtx_48 = str_repeat("a-very-long-break-string-to-clobber-the-heap", 8);
$var_vtx_18 = 0;
$var_vtx_50 = str_repeat("747 X ", 58);
$var_vtx_51 = hash_init('crc32b', HASH_HMAC, '123456');
$var_vtx_52 = str_repeat("747 X ", 58);
$var_vtx_53 = str_repeat("747 X ", 58);
$var_vtx_54 = str_repeat("747 X ", 58);
$var_vtx_55 = imagecreatetruecolor(96,51);
```

High Level Algorithm

1. Discover how to interact with the allocator via the API of the program to produce a set of API calls which can be used for layout manipulation, Z
2. Randomly combine a series of API calls from Z and provide them to the target program, including the API call to allocate S and D
3. **Check whether `addressof(S) - addressof(D) = X`, if not go to step 2, but if it is then exit and report the discovered sequence of API calls to the user**
    - Execute the generated program in an instrumented version of PHP which reports the distance

Architecture



Example – CVE-2016-7126

```php
<?php

$img = imagecreatetruecolor(10, 10);
imagetruecolortopalette($img, false,
        PHP_INT_MAX / 8);
?>
```

Example – CVE-2016-7126

```php
<?php

$img = imagecreatetruecolor(10, 10);
imagetruecolortopalette($img, false,
        PHP_INT_MAX / 8);
?>
```

```php
<?php
$img = imagecreatetruecolor(10, 10);

#X-SHRIKE HEAP-MANIP
#X-SHRIKE RECORD-ALLOC 0 1
$dst = imagecreate(1, 1);

#X-SHRIKE HEAP-MANIP
#X-SHRIKE RECORD-ALLOC 48 2
imagetruecolortopalette($img, false,
        PHP_INT_MAX / 8);

#X-SHRIKE REQUIRE-DISTANCE 1 2 0
?>
```

Example – CVE-2016-7126

```php
<?php

$img = imagecreatetruecolor(10, 10);
imagetruecolortopalette($img, false,
        PHP_INT_MAX / 8);
?>
```

```php
<?php
$img = imagecreatetruecolor(10, 10);

#X-SHRIKE HEAP-MANIP
#X-SHRIKE RECORD-ALLOC 0 1
$dst = imagecreate(1, 1);

#X-SHRIKE HEAP-MANIP
#X-SHRIKE RECORD-ALLOC 48 2
imagetruecolortopalette($img, false,
        PHP_INT_MAX / 8);

#X-SHRIKE REQUIRE-DISTANCE 1 2 0
?>
```

```php
<?php
$img = imagecreatetruecolor(10, 10);

$var_vtx_0 = str_repeat("\xf4", 8);
$var_vtx_1 = str_repeat("AAAA", 16);
$var_vtx_2 = str_repeat("747 X ", 58);
$var_vtx_3 = imagecreatetruecolor(256, 256);
<...>
$dst = imagecreate(1, 1);
<...>
$var_vtx_18 = 0;
$var_vtx_23 = str_repeat("AAAA", 8);
$var_vtx_24 = hash_init("md5");
$var_vtx_25 = hash_init("md5");
<...>

imagetruecolortopalette($img, false,
        PHP_INT_MAX / 8);
?>
```

Evaluation

- 3 vulnerabilities x 10 target data structures = 30 experiments
  - Max run time: 12 hours
  - 40 concurrent analysis processes (algorithm is trivially parallelised =))
- 21/30 (70%) success rate
  - Average time: 9m30s, Min. time: < 1s, Max. time: 1h10m
  - Average number of candidates before success: 720k
- Of the 9 failures, 8 involve a single vulnerability in which the source buffer has a size with the property that there are no fragments in the database which allocate or free a single buffer of that size

So, how does this help an exploit developer?

- With a high degree of success random search and a GA can achieve desirable physical heap layouts
- However, a correct physical layout does not an exploit make
- How can we integrate this capability into the exploit development process?

Exploit Sketching

- In automated program generation there is the concept of ‘sketching’
  - Automatically generating full programs to solve a problem is really difficult
  - Instead, the programmer sketches a partial solution and uses an algorithm to figure out parts that are difficult for the programmer but ‘easy’ for a machine
- Exploit sketching is inspired by this concept
  - Developer writes exploit sketch that describes how to trigger vulnerabilities, what data structures should be corrupted etc.
  - Machine uses the search algorithm previously described to build an exploit that ensures the correct heap layouts are achieved

Example Sketch

```
echo "[+] Forging function pointer table ...";
$ptr_table_id = dve_alloc_buffer(40);
dve_write_to_buffer($ptr_table_id,
        "EEEEEEEE" .            # 0
        "FFFFFFFF" .            # 8
        "GGGGGGGG" .            # 16
        "HHHHHHHH" .            # 24
        $shellcode_addr         # 32
);

echo " done\n";

echo "[+] Leaking function pointer table address ...";
#X-SHRIKE HEAP-MANIP 128
#X-SHRIKE RECORD-ALLOC 0 4
$ptr_table_container_id = dve_alloc_buffer(128);
dve_store_buffer_address($ptr_table_container_id, 0, $ptr_table_id);

#X-SHRIKE HEAP-MANIP 128
#X-SHRIKE RECORD-ALLOC 0 5
$oob_read_src_id2 = dve_alloc_buffer(128);

#X-SHRIKE REQUIRE-DISTANCE 4 5 128

echo " done\n";

$table_addr = dve_read_from_buffer($oob_read_src_id2, 128, 8);
$ptr_as_str = "";
$prefix = 1;
for ($i = 7; $i >= 0; $i--) {
        $v = ord($table_addr[$i]);
        if (!$v && $prefix) {
                // Leading 0
                continue;
        }
        $prefix = 0;
```

Example Sketch

```
echo "[+] Forging function pointer table ...";
$ptr_table_id = dve_alloc_buffer(40);
dve_write_to_buffer($ptr_table_id,
        "EEEEEEEE" .            # 0
        "FFFFFFFF" .            # 8
        "GGGGGGGG" .            # 16
        "HHHHHHHH" .            # 24
        $shellcode_addr         # 32
);

echo " done\n";

echo "[+] Leaking function pointer table address ...";
#X-SHRIKE HEAP-MANIP 128
#X-SHRIKE RECORD-ALLOC 0 4
$ptr_table_container_id = dve_alloc_buffer(128);
dve_store_buffer_address($ptr_table_container_id, 0, $ptr_table_id);

#X-SHRIKE HEAP-MANIP 128
#X-SHRIKE RECORD-ALLOC 0 5
$oob_read_src_id2 = dve_alloc_buffer(128);

#X-SHRIKE REQUIRE-DISTANCE 4 5 128

echo " done\n";

$table_addr = dve_read_from_buffer($oob_read_src_id2, 128, 8);
$ptr_as_str = "";
$prefix = 1;
for ($i = 7; $i >= 0; $i--) {
        $v = ord($table_addr[$i]);
        if (!$v && $prefix) {
                // Leading 0
                continue;
        }
        $prefix = 0;
```

Completed Sketch

```
echo "[+] Leaking function pointer table address ...";

$var_vtx_0 = str_repeat("\13", 91);
$var_vtx_1 = str_repeat("\13", 91);
$var_vtx_2 = str_repeat("30", 46);
$var_vtx_3 = str_repeat("\28", 48);
$var_vtx_4 = str_repeat("\13", 91);
<...>
$var_vtx_311 = str_repeat("47", 47);

shrike_record_alloc(0, 4);
$ptr_table_container_id = dve_alloc_buffer(128);
dve_store_buffer_address($ptr_table_container_id, 0, $ptr_table_id);

$var_vtx_0 = str_repeat("47", 47);
$var_vtx_1 = str_repeat("\28", 48);
$var_vtx_2 = str_repeat("\x552", 45);
$var_vtx_3 = str_repeat("\28", 48);
$var_vtx_4 = str_repeat("30", 46);
<...>
$var_vtx_216 = str_repeat("\x552", 45);

shrike_record_alloc(0, 5);
$oob_read_src_id2 = dve_alloc_buffer(128);


$distance = shrike_get_distance(4, 5);
if ($distance != 128) {
    exit("Invalid layout. Distance: $distance\n");
}

echo " done\n";
```

Demo

- CVE-2013-2110
  - Out-of-bounds write vulnerability in PHP
  - Allows us to write a NULL byte immediately after a buffer
- Exploitation strategy
  - Allocate a gdImage structure immediately after the overflow source
    - Contains a pointer to an array of pointers as its first element. If you control the array of pointers, you can use image manipulation functions to read/write arbitrary memory
  - Use the NULL byte write to change where the gdImage thinks its array of pointers is
  - Allocate a buffer we control into the space the gdImage is now using as its array of pointers
  - Standard from there – info leak, corrupt function pointer, win

Demo

- CVE-2013-2110
  - Out-of-bounds write vulnerability in PHP
  - Allows us to write a NULL byte immediately after a buffer
- Exploitation strategy
  - **Allocate a gdImage structure immediately after the overflow source**
    - Contains a pointer to an array of pointers as its first element. If you control the array of pointers, you can use image manipulation functions to read/write arbitrary memory
  - Use the NULL byte write to change where the gdImage thinks its array of pointers is
  - Allocate a buffer we control into the space the gdImage is now using as its array of pointers
  - Standard from there – info leak, corrupt function pointer, win

Automatically Completing a Partial Exploit

- See https://youtu.be/MOOvhckRoww

Conclusion

Takeaways

- Two approaches to solving the heap layout problem
  - Random search - an effective, but simple, approach
  - A multi-objective GA that on average solves over 90% of the benchmarks, and over 70% of those never solved by random search
- A method for discovering how to interact with a target's allocator via its API, based on dynamic analysis of fuzzed regression tests
- Exploit sketching – An idea that allows us to combine human creativity and expert knowledge with scalable, focused algorithms
  - New approach to the automatic exploit generation (AEG) problem
  - Allows for automation and manual effort to be deployed where they are most appropriate and traded off against each other
  - Exploitation is programming - likely more we can learn from the program synthesis community on how to embed automation into the exploit development process

Thanks / Questions?

Source code will be available around the end of August (email me now and I'll send you a link to it when it's released)

For the full **paper** on this (except the GA stuff):
https://arxiv.org/abs/1804.08470

(Or Google "Automatic Heap Layout for Exploitation")

@seanhn / sean@vertex.re