Attacks on uninitialized local variables

Halvar Flake – Head of Research
SABRE Security GmbH
halvar.flake@sabre-security.com

Black Hat Federal 2006

Introduction

Abusing memory-corruption issues in order to compromise systems has a long history:

- Stack overflows abused since the 70's in various circles
- Public usage since the late 80's
- Heap overflows abused publically since around 2000, probably exploited earlier without public documentation
- More complex issues (double-free()'s etc) published since 2002

- Remediation focuses a lot on published exploit techniques
- Various countermeasures (stack & heap canaries, front/backlink checks) proposed & implemented

Introduction

Failure to initialize local variables is more common than most people think.

- Hardly any public discussion of exploitation methods
- (correction: Since late 2005 there is a paper dealing with a specific instance under http://www.felinemenace.org/mercy)

  Public discussion seems to imply that exploitation is hard as the memory content of non-initialized memory is random or hard to control

Key points of this presentation:

- The contents of uninitialized local variables on the stack cases well-defined by the program that is running
- An attacker can attempt to determine paths that allow him to control these values
- Success in controlling the values will allow compromise

Warning

This talk is work-in-progress

- My first approach to the problem will be presented
  - It was fairly useful in practice
  - But it suffers from severe problems
- My second approach to the problem will be presented
  - It is more accurate
  - It still suffers from problems, but fewer
- The discussed ideas are far from perfect
- It is often surprising how much 'wiggle-room' the complexity of the application leaves for an attacker
- Yes, there are quite a number of instances where non-initialized variables are not controllable. In that case, you will have to go fishing again

Some questions

How can we talk about a 'fish-class' or 'bug-class' in general ?

- In many situations we do not have a large number of specimen at hand
- Every instance of a bug-class is often subtly different
- Generic methods usually emerge when lots of different fish of the same species have been caught and prepared for consumption

Now we're looking at a new 'species of fish' – how do we learn how to prepare it if we only have one ?

Can we 'breed' fish for practice ?

Problems

Artificially creating fish has to be done with care – if we do it wrong, we will end up with different fish than what we would find in the wild.

- Manually created sample applications will hardly ever mirror complexity of real-world programs
- Creating sample apps with certain bugs is hard to do in a manner that is unbiased
- Perhabs a better approach: Take an arbitrary function that could exhibit such a problem in an arbitrary application and introduce the flaw there. Then think about exploitation methods

We might have 'created' a fairly realistic approximation of the 'real thing', and can study how to make use of it.

Doesn't the compiler warn me ?

Compilers will warn programmers about the failure to initialize local variables in many cases, but ...

- Compilers do not do interprocedural analysis
- Because of different compilation/linking situations, interprocedural checking isn't practical in many build situations
- If a pointer to a local variable is passed to a subfunction, the compiler considers this local variable to be initialized by the subfunction

Let's have two examples to clarify:

Compiler warns

The compiler will warn in a case like this:

```
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv )
{
     int b;
     printf( "%lx", b );
}
```

Compiler doesn't warn

The compiler won't warn in a case like this:

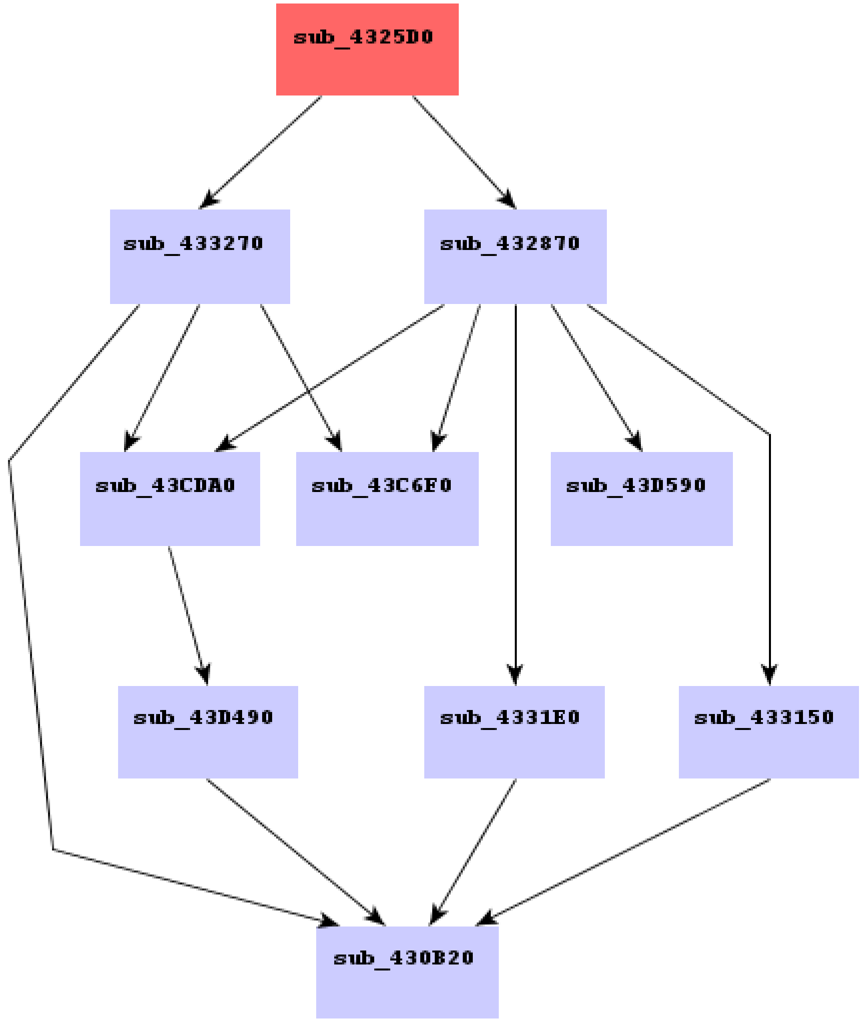
```
#include <stdio.h>
#include <stdlib.h>

void take_ptr( int *bptr )
{
    print( “%lx”, *bptr );
}

int main( int argc, char **argv )
{
    int b;
    take_ptr( &b );
    print( “%lx”, b );
}
```

What is the scenario ?

We're looking at the following situation then:

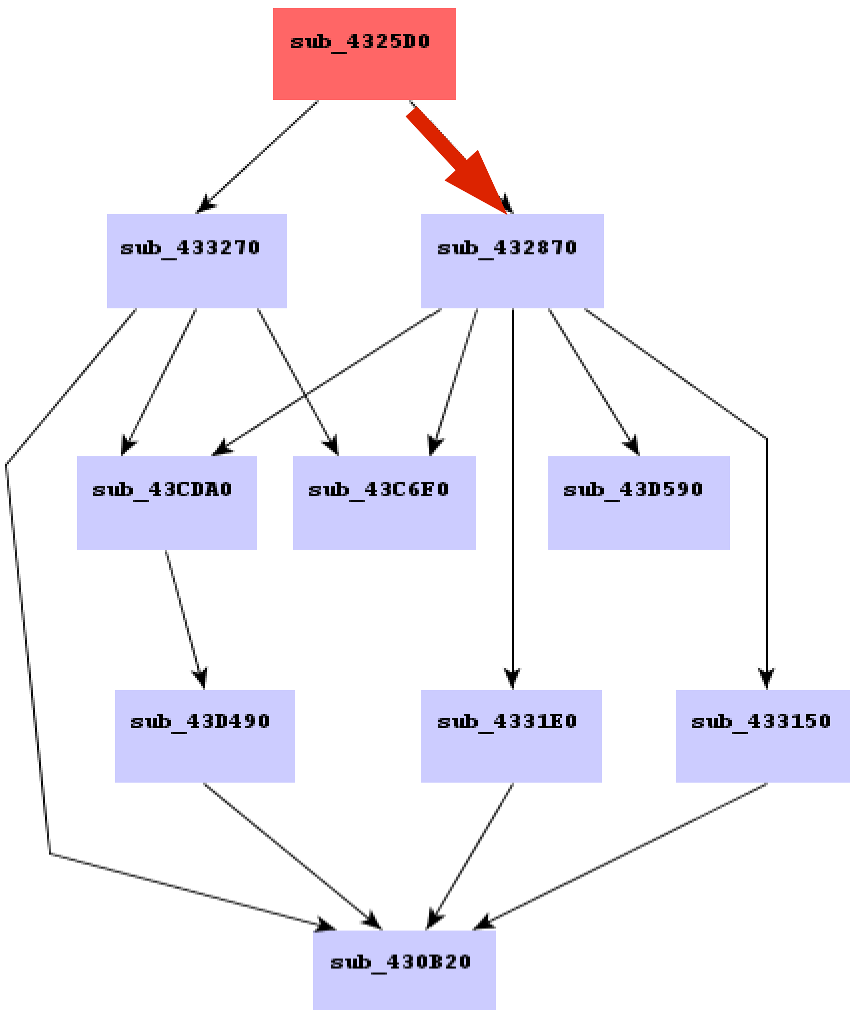
- Application uses some sort of data structure on the stack (including regular variables)
- Application calls a subfunction to initialize the data structure or variable
- Attacker can somehow make that subfunction fail
- Application does not check for success of that subfunction
- Further assumptions:
  - Attacker has input to trigger the issue

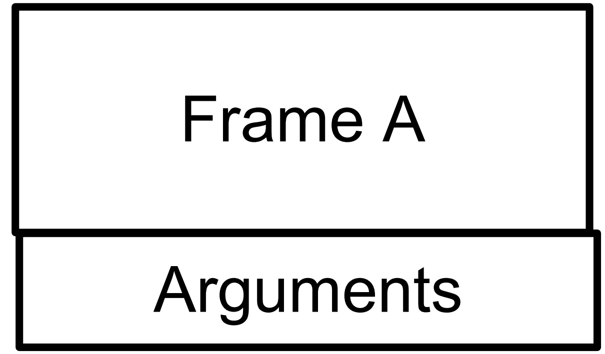
Let's look at the stack ...



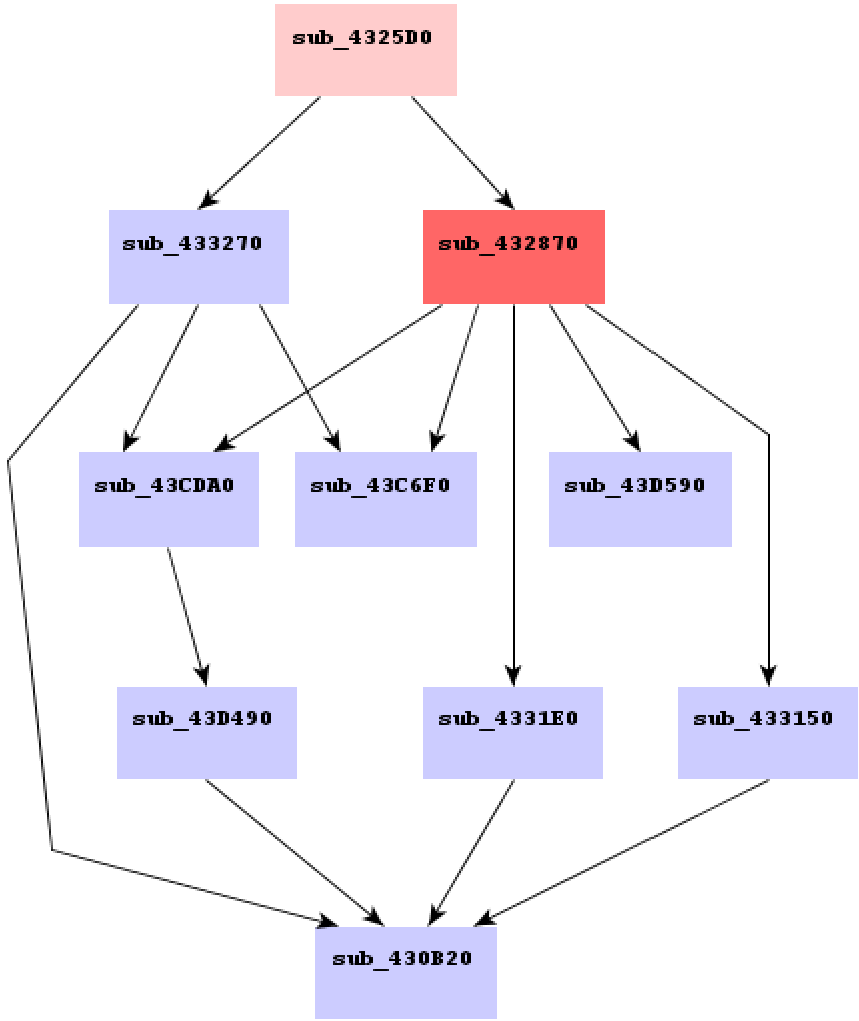
Frame A

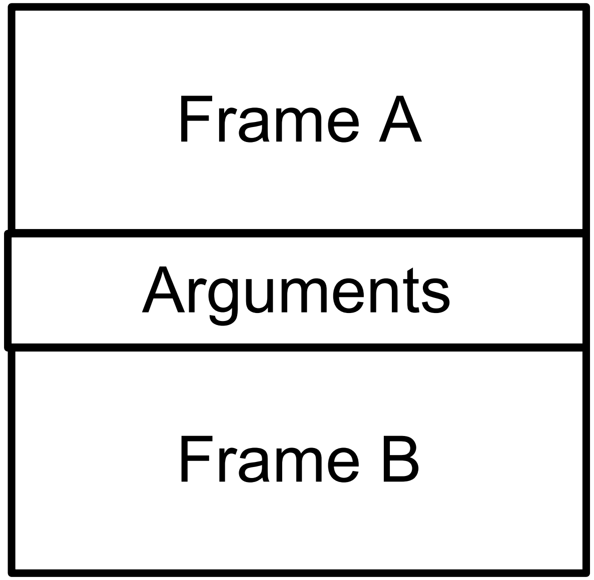
Let's look at the stack ...



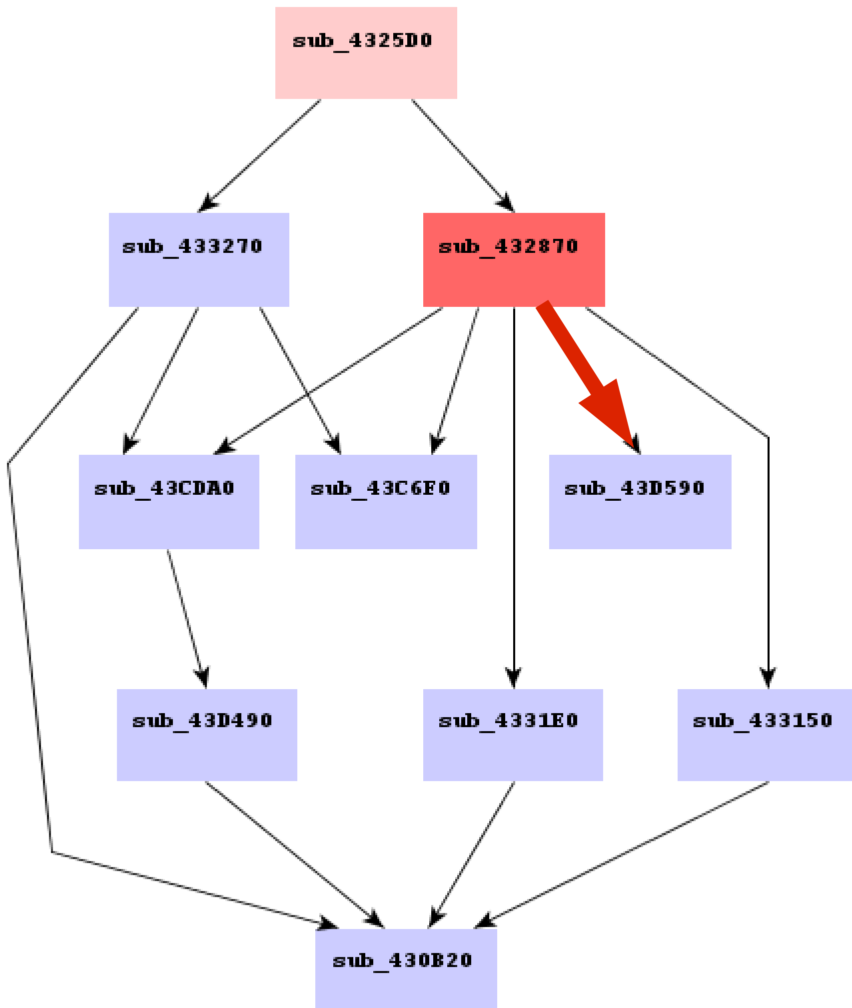


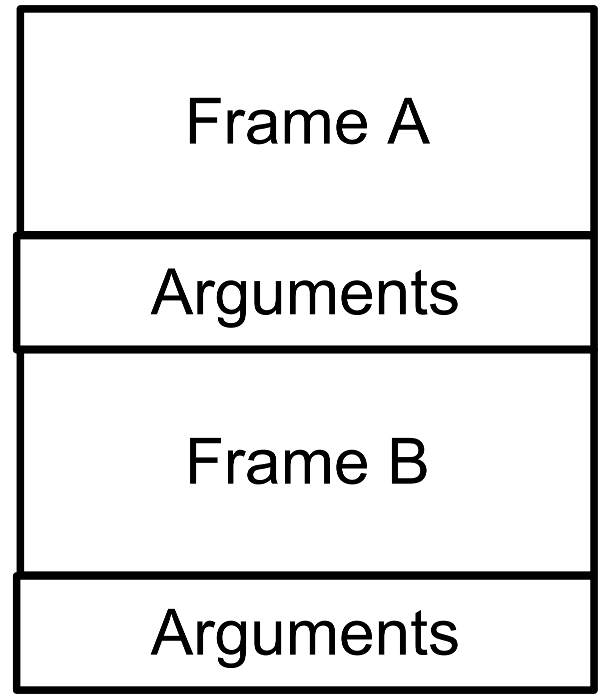
Let's look at the stack ...



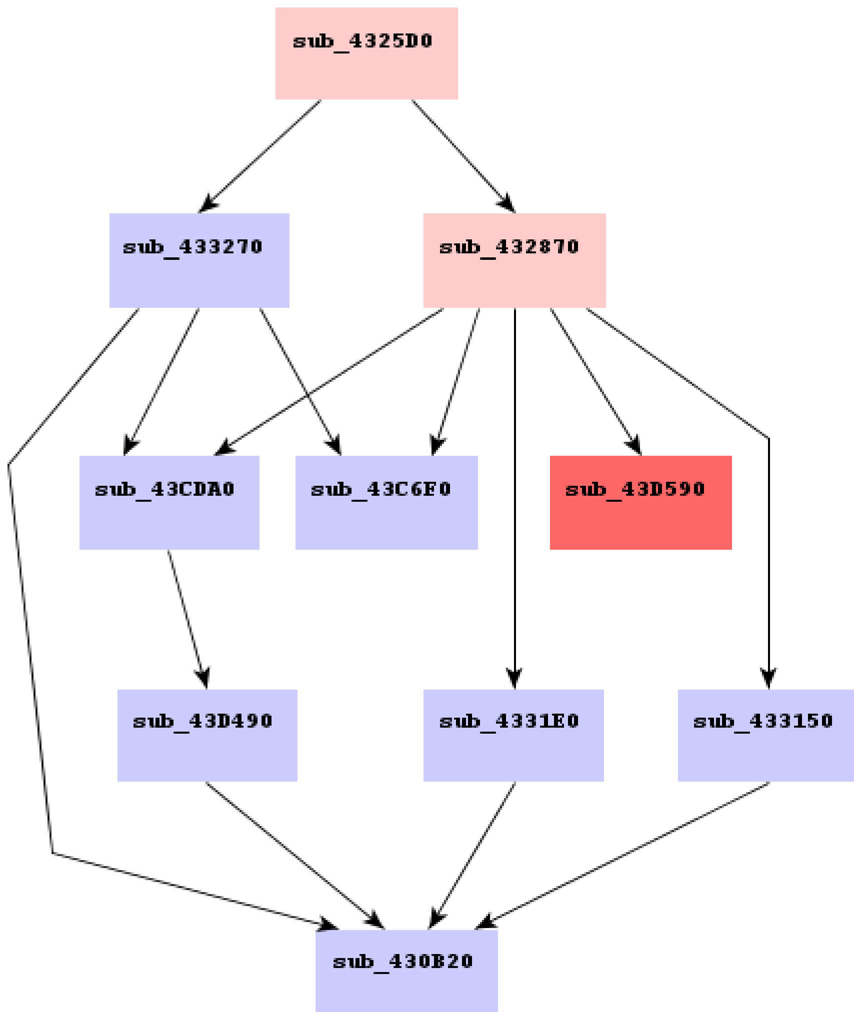


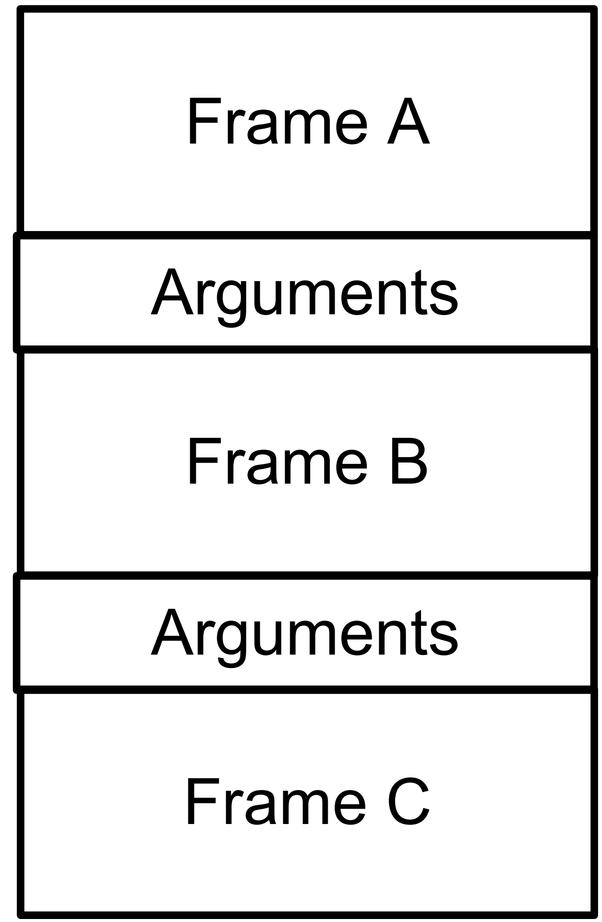
Let's look at the stack ...



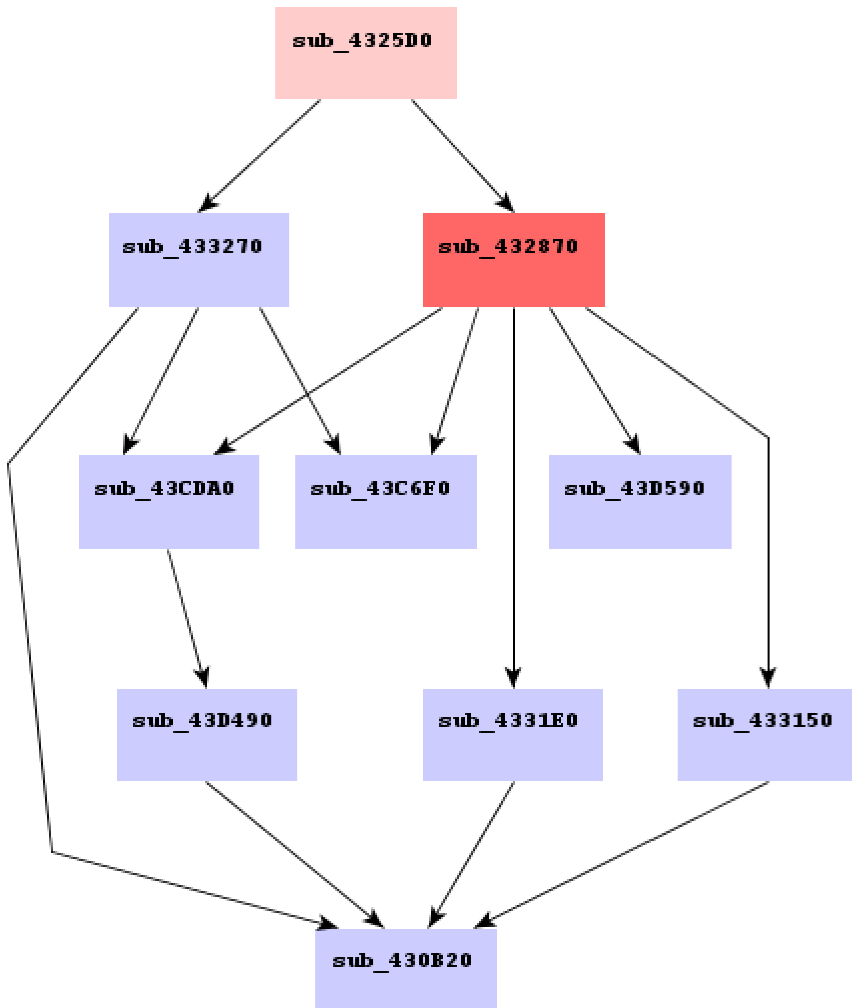


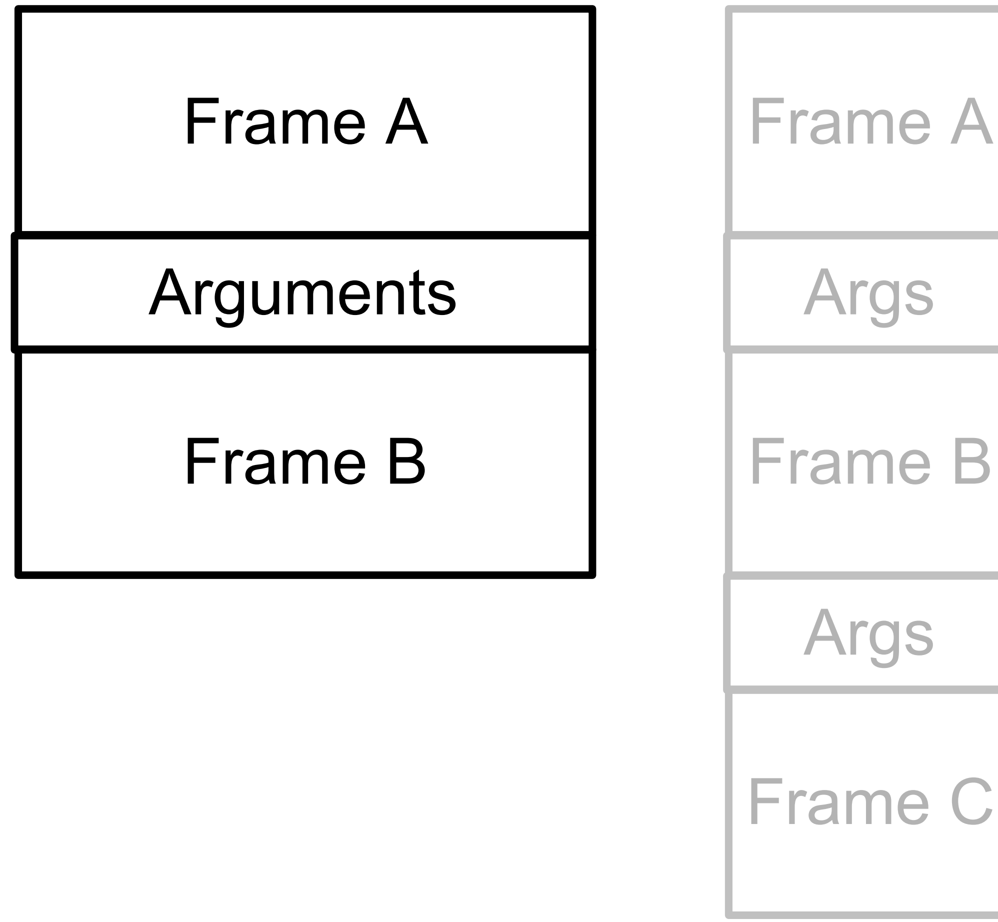
Let's look at the stack ...



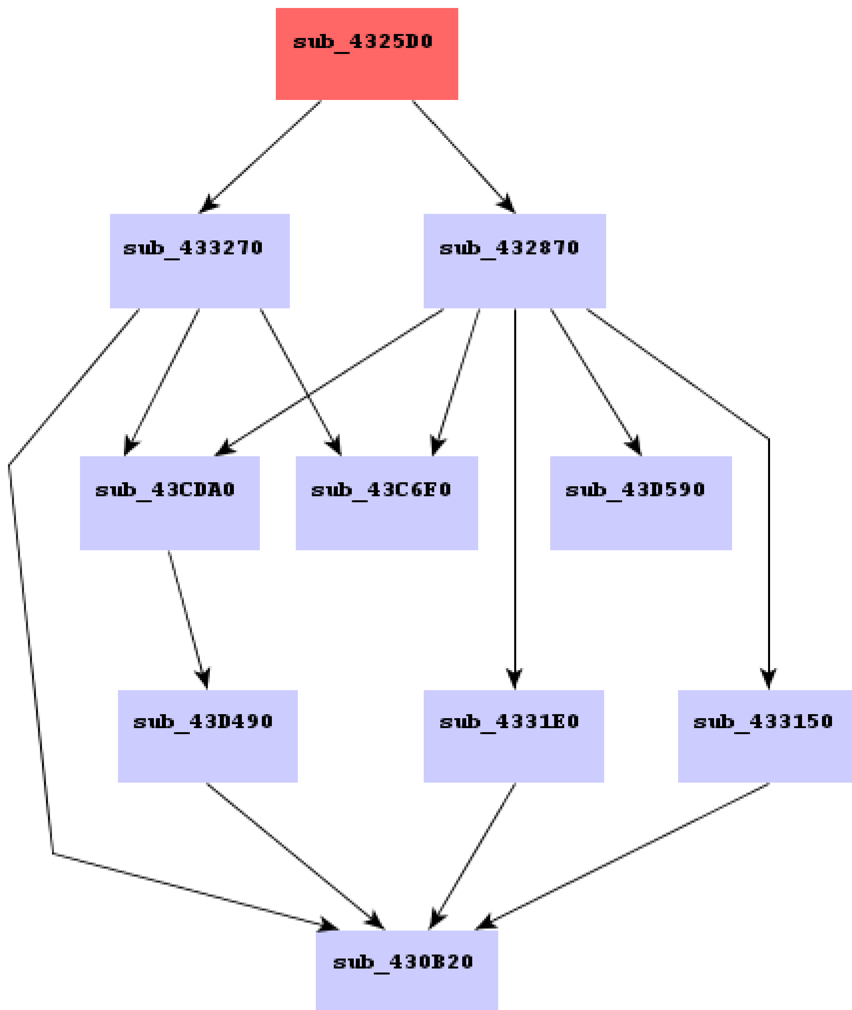


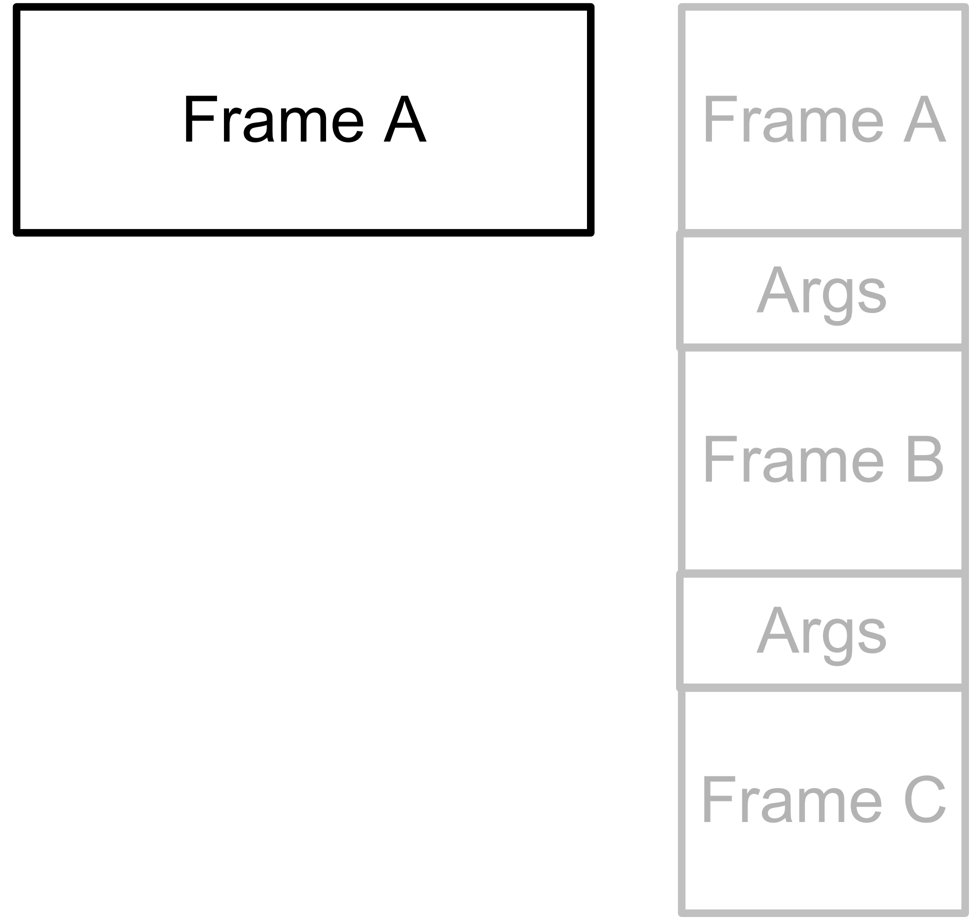
Let's look at the stack ...



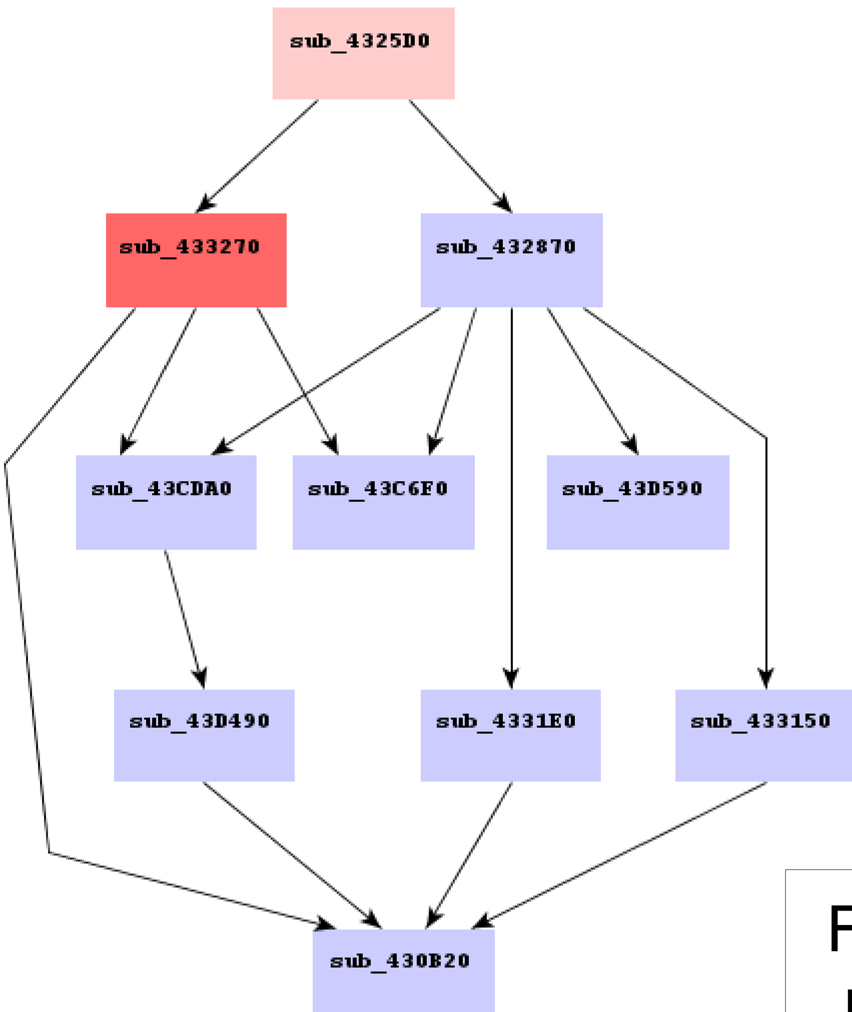


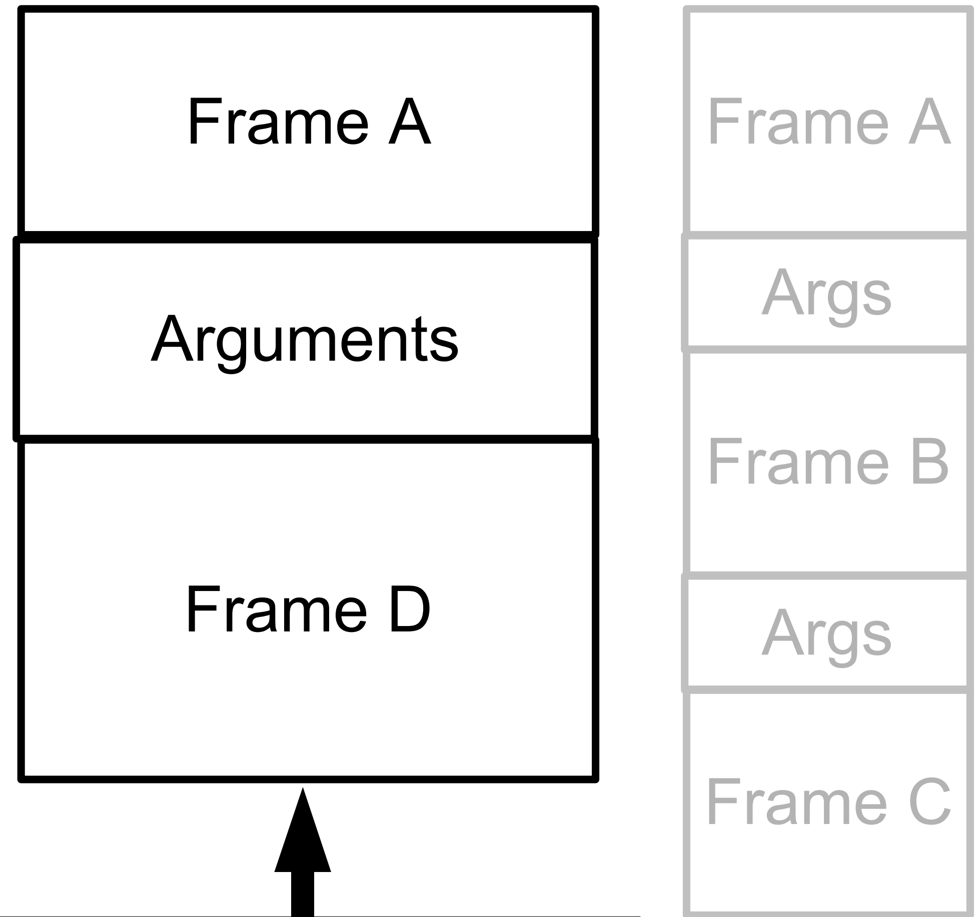
Let's look at the stack ...





Let's look at the stack ...





Frame of D overlaps with parts of the B, C frames and with some arguments

So what do we do ?

We need to “initialize” the stack variables ourselves to make use of them

- Identify which other program paths could access the memory that ends up being used
- Choose one that allows attacker-supplied data to be written to those memory locations
- Craft input to exercise this program path
- Exercise the vulnerable program path without 'clobbering' the data that we wrote again
- Have fun using pointers and data that we supplied

Approach #1: Delta-Graphs

The following slides will explain my first attempt to deal with the problem. It has many severe flaws, but served well in a few situations.

The silly name comes from the fact that we are building graphs annotated with stack-delta's.

Local vs. Global analysis

- Ideally, we should look at the program in it's entirety – consider the whole program
- Problem: The number of possible paths through the program is exponential in the number of functions
- Problem: Most algorithms in code analysis are O(n^2) or worse
- Cop-out: Instead of looking at the entire program, we only consider a small subset that might be interesting for us.
- Reason: We can always increase the scope of the analysis if we fail with the 'restricted' scope

Local vs. Global analysis

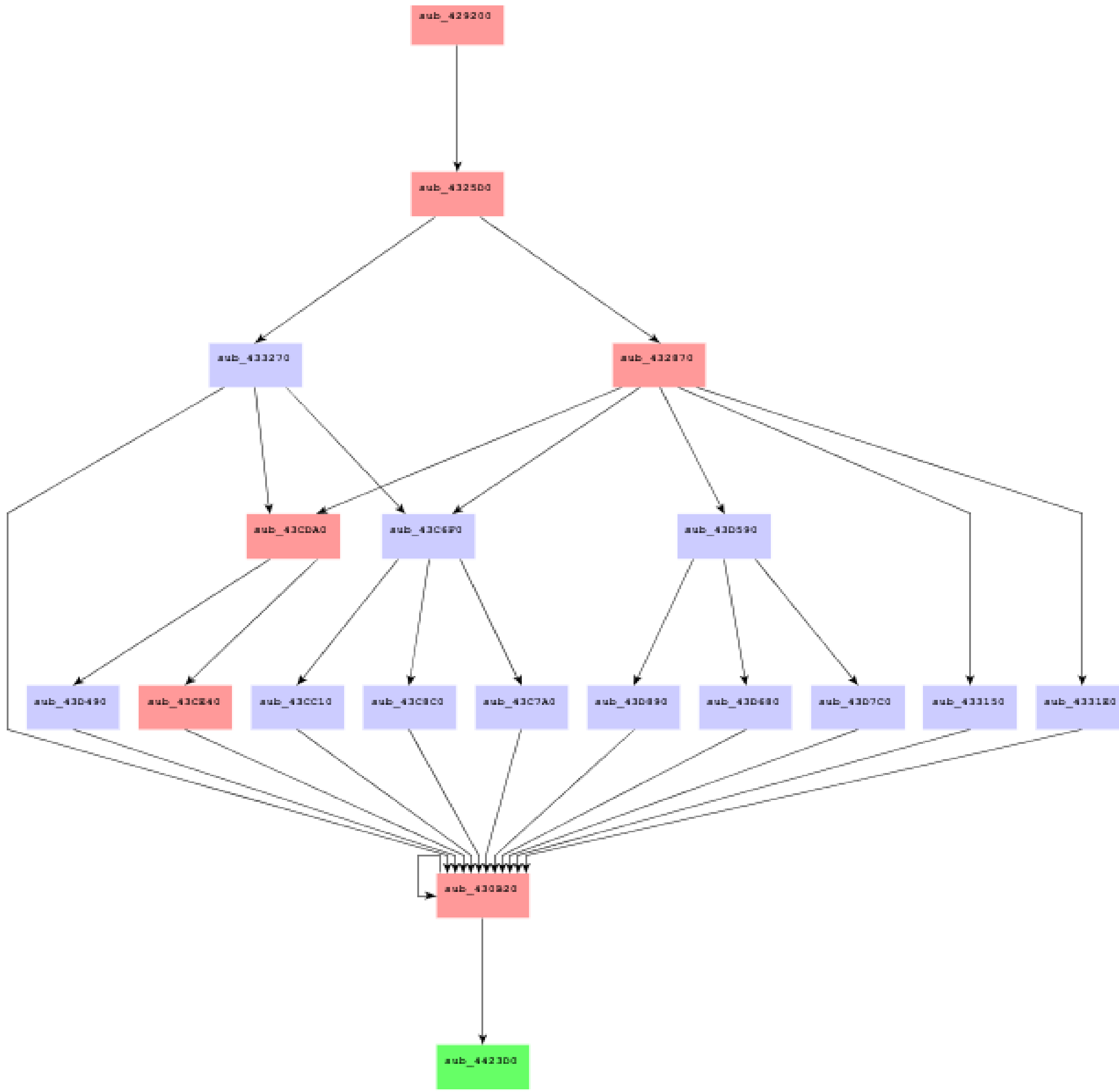
Let's create some terminology:

- “Init Path” -- the path that we are going to use to write the data
- “Trigger Path” -- the path that is going to use the data

It is in our best interest if the “Init Path” is very close to the “Trigger Path”:

- We need to build input for the “Init Path”, which is time-consuming
- If we take a drastically different path we increase the risk of accidentally clobbering our data again (more on this later)

Local vs. Global analysis



Local vs. Global analysis

We will work locally, but parametrized: Only “Init Paths” which diverge from the “Trigger Path” only on the last “n” steps will be considered.

- We can start with small “n” (2, 3) and expand if we need
- In simpler cases we can actually see & understand everything

A “stack-delta”-graph

We walk back the chain n steps (let's take 2 for now)

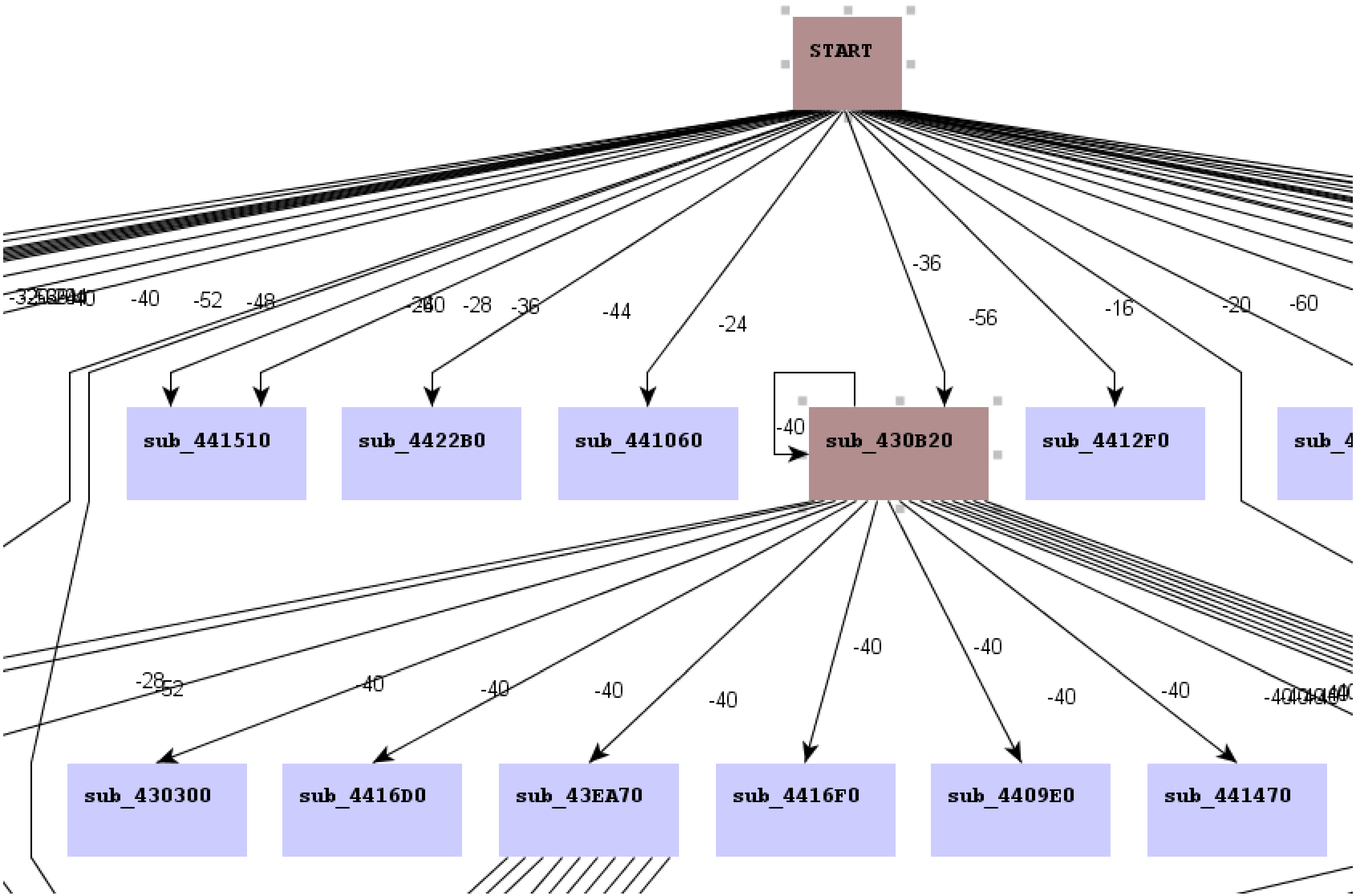
From this point onwards, we generate a callgraph of all functions.

Each edge in the graph represents a “call”

Each call has a 'stack delta', specifically the change to ESP done in this function before the call

We annotate each edge in the graph with that number

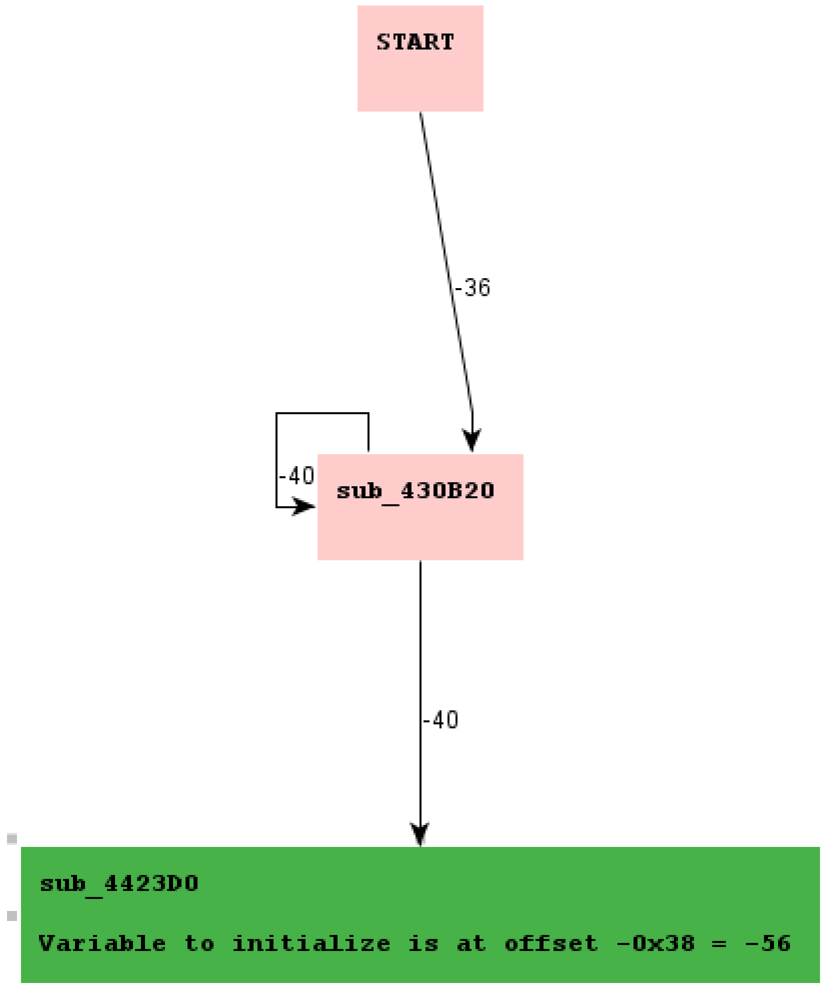
A “stack-delta”-graph



Calculating the distance

We now calculate the distance the stack variable we want to initialize has from ESP upon entry to our chosen “START”:

-56
-40
-36
-04
-04
=-140



A “reachability”-graph

We now start to explore all paths through the “stack-delta”-graph.

- Explore graph in depth-first-search
- On each edge, keep track of the stack delta accumulated at this point
- Each time a function can be reached with a different stack delta, it receives a separate node in the graph
- Normally, this graph would be exponential in size to the “stack-delta”-graph
- We limit our search: If the accumulated delta is already lower than the distance we calculated, we stop

A “reachability”-graph

The resulting graph gives us a number of functions which can have “overlapping” stack frames with our target variable.

This is nice and a good point to start, but the generated graph suffers from a severe problem:

- Problem: The graph has no sense of “order” -- if one of the calls on our “path” happens at the beginning of a function, this will lead to a large number of false positives

So for better results we will need a better graph

Illustration of the problem

Order can end up being quite important.

Approach #2 (I)

The first approach obviously abstracted too far. A second approach will have to stay closer to the assembly code. A short overview of what we're going to do then:

- Use the path that we already know how to exercise
- Take the flowgraphs of the functions in this path and 'glue' them together in a sensible manner
- Inline all called functions into the resulting graph
- Annotate each basic block with the change it imposes on the stack pointer
- Create a 'reachgraph' – traverse the graph upwards
- We get a graph that shows us basic blocks that might access our memory

Approach #2 (II)

Illustration of what we are going to do:

- Decide on a path through the callgraph – we take the one we already use. It ends up just being a linked list:
- From each flowgraph, cut the nodes that will not lead to where we want to go
- Add edges from the “call” instructions to their call targets
- Resulting graph shows all possible paths to the target function using the sequence of functions from our original path
- Pretty output is not yet available, so back to VCG :-(

Approach #2 (III)

The resulting graph is weakly ordered by stack-depth – nodes are “deeper” on the stack by being further away from the beginning of the graph.

Therefore nothing in this graph overlaps with our target stack-frame.

We now inline all subfunction calls in this graph several steps deep (if possible all the way).

The resulting graphs can be quite large.

Approach #2 (III)

We associate each basic block with the change it imposes on the stack pointer

We then work similarly to the “reachgraph”: Traverse the graph upwards, accumulating the delta's on each step – if the delta ever drops below zero, we have a function that overlaps.

The resulting graphs

We get 2 choices 'nearby', but also 2 more quite far removed.

How about more results?

- We stuck directly to the path that we were already taking
- We only inlined 2 function layers deep
- Inlining more deeply will give us more liberty
- Allowing more variation along the path will give us more liberty: Instead of considering only paths that follow the calltree path that we recorded, we can consider all paths between two points
- Careful: The further we move away from paths that we exercise, the more prone we are to choosing logically inconsistent paths

What's next ?

- The current algorithm only determines functions that have overlapping stack frames
- The next improvement should be: Determining basic blocks that write to our variables
- Problem: We might end up with aliasing issues
- What about research on uninitialized heap variables ?

Other limitations ?

- Most of this was developed on embedded targets
- No handling of C++ indirection: All dynamic calls have to be resolved in the disassembly
- No handling of external libraries – you will have to load all relevant DLLs into your IDB along with the application on windows
- All code x86-centric at the moment

Tools used

- Datarescue's IDA Pro Disassembler as disassembly engine: http://www.datarescue.com/ida
- SABRE BinNavi for graph visualisation & recording of program traces: http://www.sabre-security.com/products/BinNavi.html
- IDAPython for scripting http://www.d-dome.net/idapython
- A home-brew IDAPython library that provides more comfortable access to flowgraphs, inlining etc. http://www.sabre-security.com/x86_RE_lib.zip
- Warning: The above library is experimental code without documentation. Using or reading it can be detrimental to your health.

Questions ?

- I probably have at least as many questions that I can't answer yet as the audience
- Practical experience: This stuff works surprisingly well
- Being able to initialize program pointers directly bypasses heap / stack canaries