About stack variables recognition and how to thwart it

Seekbytes

1 Introduction to local variable inference

The secret art of reverse engineering is an imprecise one, built on tools that rely on very advanced techniques to be able to reconstruct the high-level code from a given executable file. Given any instruction set architecture (known examples: Intel, ARM or JVM), the real challenge is to recover the set of high-level elements that the compilation has removed or transformed. Example of what the compiler removes may include: variable names, flow control constructs, strings, and in general all the high-level details not needed at the low level.

In this article, I would like to talk about how most decompilers manage to infer about the allocation of local variables within the function. As soon as the disassembly phase, which involves transforming bytes into understandable instructions, is completed, the decompiler begins its work by applying a series of fixed-point analyses, such as dataflow analysis (constant propagation, liveness), and the time comes when it must try to reconstruct the local variables of a function. That is, figuring out which variables the low-level code uses are allocated on the stack, destroyed as soon as the subprocedure call returns the value. The stack is in fact used for three main purposes: to pass arguments from function callee to function called, to allocate temporary variables that are valid only for the scope of the function, and to store the return address that is retrieved when a return statement is encountered within the function.

2 The semi-naive algorithm

The most common decompilers – including Binary Ninja, IDA, and Ghidra – may use a very naive version of the variable retrieval algorithm that works based on index within the `base pointer` register, also called BP-frame heuristic. It is much easier to write an example than to explain it: within the disassembly code, we have several mentions of the base stack pointer. Instructions of the type `mov [ebp-0x14], eax` are actually converted to simple `MEM[ebp-0x14] = eax` by an operation called lifting. This allows the decompiler to be able to immediately say that what is pointed to the address of ebp minus `0x14` must take the contents of the general register eax. When we find `ebp - 0x14` or `esp - 0x14`, we are most likely referring to a local variable at address `-0x14` named in most cases `var_14`.

The decompiler recognizes that address in memory and assumes that a new variable has been declared within the "hypothetical" high-level source code. The decompiler runs through the entire instruction set of the basic blocks on which a function is built on. The algorithm keeps track of all the accesses on the stack: for each new index it encounters on the stack, it allocates a new variable whose type it does not yet know but knows that there is a write/read at that address. The stack analysis algorithm is thus completed by going to check where parameters are saved and further checks are needed to ensure the analysis is sound (such as stack balancement, and checking if there are any overlapping variables).

3 Consequences of using the semi-naive algorithm

The consequences of using the semi-naive algorithm are that you can freely manipulate the base pointer of the stack so that you can trick the decompiler into thinking that there are variables or arguments where in fact there are none. The decompilers build part of the later stages on the fact that they can have a crystal-clear view of the stack and somehow infer elements on the stack because **by default** speaking about the size, the stack should be no larger than 1MB, allowing a fast analysis.

However, the semi-naive algorithm can be easily exploited by some actors whose goal is to break the decompilation. The trick is to make the decompiler believe that it is using a very large area of the stack, when in fact that area is only the result of the decompiler's overapproximation. The overapproximation comes from considering each branch as alive (i.e., the program could jump at runtime even to the one considered dead). While in reality, experience says that someone may create potential dead branches that will not be executed at run time, and thus the context created does not take consideration of that branch.

To successfully inject an impossible index for the stack variables:

1. create a dead execution branch that will never be executed at runtime (e.g., via conditions that we know are *a priori*, always true or always false via *opaque predicates*).
2. mention access to a local variable placed in a very high or very low value of the stack base pointer in the branch never executed.

Note that this also works for values that go above the base of the stack pointer, i.e., the arguments (much also depends on how the stack is constructed, whether upward or downward). In addition to destroying local variables recognition, it is possible to cause the decompiler to make very different assumptions about how arguments are passed at runtime, making it almost impossible to recognize the usual call conventions.

```
#define huge_sp_predicate_for_local_variables \
__asm__ ("push rax \n"\          // mem[sp] = rax
         "xor eax, eax \n"\      // eax = eax ^ eax
         "jz live_branch\n"\     // is eax == 0?
         "and ecx, [rbp - 123456] \n" // huge value
         "live_branch: \n"\      // always true branch
         "pop rax\n");         // rax = mem[sp]

#define huge_sp_predicate_for_arguments \
__asm__ ("push rbx \n"\              // mem[sp] = rax
         "xor ebx, ebx \n"\          // eax = eax ^ eax
         "jz live_branch_2\n"\       // is eax == 0?
         "and ecx, [rbp + 123456] \n" // huge value
         "live_branch_2: \n"\        // always true branch
         "pop rbx\n");             // rax = mem[sp]

// where needed
huge_sp_predicate_for_local_variables;
huge_sp_predicate_for_arguments;
```

If you recognized the dead branch, fixing it is very simple for an analyst in IDA: you can click the portion of the assembly code that you think is dead and with right-click use the `Undefine` option. Another alternative is to manually edit the stack via the `Edit function` option, or change the heuristic for finding the local variables.

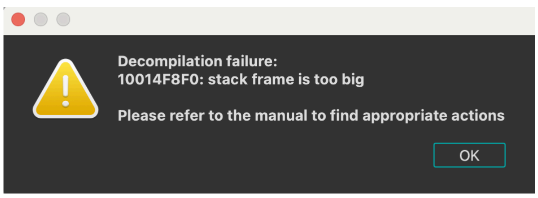

Figure 1: The final result

If you are looking for a good way to dumbly destroy the inference of the IDA decompiler, perhaps forcing people to look directly at the assembly code and not start with the decompiler, you could use this technique. This transformation has been used a lot in obfuscating binaries such as Apple Fairplay, where the combination of local variables allocated to very large stack offsets and arguments partly destroyed static analysis. As an exercise left for the reader, someone might want to understand if this can be applied also to SP-based heuristic.