Abstract

Most software attacks are executed by exploiting vulnerabilities of the software and subverting the control flow of a certain program. Those attacks would not be possible if the control flow of a program is forced to remain within certain boundaries. Control Flow Integrity offers those guarantees by restricting the flow of the program within the boundaries of the Control Flow Graph. Furthermore, it is a simple technique which can be implemented by efficiently using software rewriting and it is effective even when the attacker has complete control of the system memory. This work aims to present the principles of the Control Flow Integrity, demonstrate a specific implementation of the CFI and finally present how it can be applied along with existing control flow techniques to strengthen and enforce further security policies.

I. CFI Instrumentation

As stated before, the principle idea of CFI is to confine the program within the boundaries that are predetermined by the Control Flow Graph. The CFG is created by static binary analysis of the programs but can also be created by source code analysis or execution profiling. The figure shows the CFG of the code on the left. CFI instrumentation modifies according to a given CFG each source instruction and each destination instruction of computed control flow transfers. This is done by inserting a unique bit pattern or ID at each destination and a dynamic ID check before each source to ensure (at runtime) that the destination has the proper ID class. In the figure below, the edges start from instruction that transfer control flow, functions calls and returns. What should be noted here is that sort() has two valid return locations, here it is not specified which is the actual caller so if sort is called we cannot know whether it is the first caller or the second caller when returning, this problem is tackled using the shadow call stack which is discussed later. Since sort is called twice in the sort2(), the ID 55 is inserted twice in the body of sort2(). Similarly because sort() can call either lt() or gt() the ID (17) for both of those functions is the same. Finally since both comparison functions return to the same callsite the ID (23) check for return is the same.

```c
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len) {
    sort(a, len, lt);
    sort(b, len, gt);
}
```

Figure 1: Example program fragment and an outline of its CFG and CFI instrumentation.
II. CFI Instrumentation Code

The figure above shows two alternative forms of IDs and ID checks implemented in x86 with their actual opcode bytes. The code without CFI at source is a jump instruction and at destination a mov instruction, ecx contains the jump address. In (a) alternative code the is modified to follow the rules of CFI by inserting a check before the jump instruction that checks the bytes of the destination and also insert a unique ID to the destination bytes. The ID to the destination is inserted using the ‘data’ instruction and as seen from the figure the destination bytes start with the 78563412 which is considered the ID of the destination. In order to check those bytes the cmp instruction at source is inserted and since ecx has the address the check can be done as shown, if this checks fails it jumps to the error label. However if it succeeds the jump address must be increased by 4 since we added a new instruction at the destination and it should not jump to the ‘data’ instruction. ‘lea’ instruction loads the address of the next instruction of ecx to ecx. In alternative (b) it is assumed that eax is available, the destination address is not modified but the destination label ID is synthesized by using the ‘prefetchnta’ instruction. Also the constant for the ID check is the ID-1 and it is incremented by 1 before comparing it to the destination label. This is done to avoid the case where the attacker can affect ecx and evade the check of alternative (a).

I. Three necessary assumptions

In order for CFI to be effective even in cases where the attacker assumes full control of the memory three assumptions must hold.

- **NXD** - Non Executable Data, if the data is executable the attacker might cause the ID labels to be executable. NXD alone thwarts attacks that try to execute data but not if that code pre-exists as ‘jump-to-libc’ attacks.

- **NWC** - Non Writable Code, if the code is writable the ID checks can simply be overwritten by the attacker.

- **UNQ** - The ID patterns that are chosen must not be present anywhere else in code memory except the checks and IDs, so the chosen IDs must not conflict with opcode.

III. Implementation

The implementation below modifies a function code, the function call and function return instruction are modified following the rules mentioned earlier and combining the two methods discussed. Here the ID is not increased at run time as it is done in the alternative (b) of the previous example, however the ‘prefetchnta’ instruction is used to form the ID of the destination.
The code above shows how the function call and return are instrumented to follow the CFI rules. The address of the function is stored in the memory location ebx+8 and the return instruction if called with the argument 10h it also pops 16 Bytes from the stack. Again the comparison is done directly without increasing the label ID, if it succeeds the call instruction is executed and the bytes after the call instruction must be given the ID for the return check. Since the return address is at the memory location where esp points to that address is loaded to ecx. Now the bytes to be popped from the stack are 4 more than before since the mov instruction was added. The comparison is the same as before and if it succeeds it will return to the caller.

IV. BUILDING ON CFI

As it was mentioned CFI can be used to enforce further security policies. IRMs (Inlined Reference Monitors) enforce security policies to subject programs by inserting certain validity checks. Thus, those validity checks must not be violated, CFI with SMAC (Software Memory Access Control) can be used to achieve this.

I. Faster SFI

SFI is a particular IRM which is ensures that memory accesses lie within a certain range. The example below how can CFI improve the performance of SFI as it removes the SFI check out of the loop. The SFI check is the ‘and’ instruction which masks the top bits of the address of the array. Without CFI the check must be done with every add instruction in the loop to prevent any instruction execution with arbitrary values in registers esi and ecx, however CFI preludes those jumps.

```c
int compute_sum(int a[], int len)
{
    int sum = 0;
    for(int i = 0; i < len; ++i) {
        sum += a[i];
    }
    return sum;
}
```

```assembly
...  
mov ecx, 0h ; int i = 0
mov esi, [esp+8] ; a[] base ptr
and esi, 20FFFFFFh ; SFI masking
LOOP: add eax, [esi+ecx*4] ; sum += a[i]
inc ecx ; ++i
cmp ecx, edx ; i < len
j1 LOOP
```

Figure 5: Leveraging CFI for optimizations: hoisting an SFI check out of a loop.
II. SMAC: Generalized SFI

SMAC can be used to enforce policies other than the traditional memory protection. For example it can create isolated memory regions that are accessible only to specific pieces of code for instance an library function. CFI can help with SMAC optimization as it is done with SFI. SMAC can remove the requirement for NXD and NWC by preventing access to those addresses.

III. A protected Shadow Call Stack

What was unclear before is that with CFI does not guarantee that the return location will be the most recent caller but it can only guarantee that it is one valid caller as it was discussed in section I. In order to guarantee that every call returns to the most recent caller a runtime stack must be maintained to keep track of those calls. Naturally, that stack should reside in a protected memory area, SMAC could be a solution to this however a simpler and more efficient solutions is used. x86 offers segments that are protected and by using the segment registers to refer to those segments the shadow stack can reside in those isolated areas. The proper use of those segment registers is guaranteed by CFI whereas without CFI for example, the opcodes for loading an improper segment selector might be found within blocks of system library code, thus the use of those registers could not be trusted.

V. Conclusion and Remarks

CFI is a powerful technique that enforces the flow of a program. This is a major advantage due to the fact that most attacks try to circumvent the flow of a program in order to execute the attacker’s code. Attacks like jump-to-libc are considered very powerful and can overcome the existing mechanisms of NXD since they use pre-existing code however if the flow is constrained within certain boundaries of a CFG those attacks are not possible. The problem of the general CFI implementation is that it cannot be guaranteed that the return to source actually returns to the most recent caller, it can only be guaranteed that it returns to one of the valid callers. This flaw can be mended by using a runtime call stack that keeps tracks of the functions calls. The control flow can be guaranteed by the use of high-level programming languages however CFI can be implemented in machine code level which makes it compatible with existing source code unlike high-level programming languages.