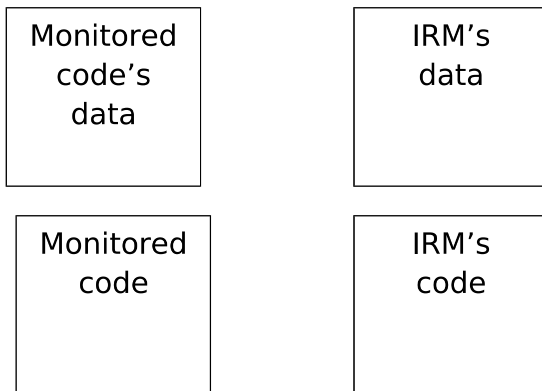
Inline Reference Monitoring Techniques

In the last lecture, we started talking about **Inline Reference Monitors**. The idea is that the policy enforcement code runs with the same address space as the code being monitored. We can roughly divide the memory region into four parts as below.



Software Fault Isolation:

One of the most important things is that we need to protect the IRM's data from being modified by the monitored code because if it can modify the IRM's data, it can evade the policy and do whatever it wants. So what we said was that there is a way to instrument the code in such a way that the memory accesses made by the monitored code can be checked at run time in such a way that we can statically prove that monitored code can only write (or may be read) only its own data. We talked about a particular technique called **Software Fault Isolation** and the way it enforces is that it ensures that IRM's data will not be accessed by the monitored code. And basically what it does is:

- Check upper bound
- Check lower bound
- Perform memory access.

So essentially, every memory access performed by the monitored code is prefixed with two instructions that are designed to check if the reference is going into the IRM memory. If not going into the area, it is allowed to do the access.

Then there is a way to circumvent this, because malicious code can jump to the instruction to perform the memory access bypassing the checks. We need a mechanism so that this bypassing cannot be done. This could be done by reserving a register such that all data accesses will use that register and then by making sure that the register will always contain a legal value before any control transfer instruction or

of course before every memory access. So any time a control transfer instruction is executed, if you can already make sure that at the point jump is performed, the register already has a valid value or a value that will check the bounds, then it is fine even if the code arbitrarily jumps there because at the point where jump is performed you already know that the conditions are being satisfied. The basic idea of SFI is to introduce the notion of a dedicated registers so that these types of checks can be put into the middle of the code you don't trust.

In addition to protecting IRM data, we need to make sure the IRM code is protected and the entry into IRM code is regulated. Presumably the code in IRM has some sort of privileged level, and so the monitored code should not be allowed to invoke arbitrary code within the IRM. So we need to protect the IRM code keeping in mind that at some point the monitored code does need to invoke the IRM code. So there has to be some way to call and the call needs to be regulated. So how would we do this?

Gates:

We can use the concept of gates that we discussed in the context of switching between privileged and unprivileged execution modes. Gates enable the callers to tell that they want to enter, but they cannot say where they want to enter, for instance. So there can be one function in the IRM code, and that is the only entry point for the monitored code and everything else that need to be passed into, should be passed as data. (Side note: You need to think about the **Time of Check** and **Time of Use** sort of issues. If the parameters are passed through registers, that will remove time of check and time of use sort of issues.) We can generalize this case to permit multiple entry points, but still, it is the IRM that needs to decide the set of entry points that can be invoked by the monitored code.

Again the technique is used to insert these checks before a jump. The jump is either within the monitored code or if it goes outside the monitored code, it goes one of the allowed entry point of the IRM code. Again there are the same issue of jumping pass the checks and here also same technique will work. But since IRM code addresses are likely to be different from data addresses, you will likely need a second dedicated register to enforce again you need different registers to tell you what is valid for data access.

In this technique, the instrumented code will be generated by the compiler. But since this is generated at the site you don't necessarily trust, on the site the code will be executed together with the site supplied IRM, you have to actually check the instrumentation. They developed ways to do that and it is not so much complicated in case of RISC instruction sets where decoding instructions and disassembling is fairly simple because you have fixed length instructions and you have instructions thar are reasonably large (say, 32 bits) and jump targets have to be aligned on 4 byte boundary. These factors make it simpler and verification can be done easily. On the other hand, in case of complex instruction sets like instructions in x86 machines, the problem becomes a little bit harder. Plus, you don't have so many registers for dedicating registers.

Control Flow Integrity:

The notion of CFI is motivated very much by the example of jumping into arbitrary code to defeat some policy. But if you design properly, you can avoid it. But the question is: **Can anyone define**

some notion of CFI which talks about some integrity property regarding control flow transfers such that using this property you can build higher level security mechanism? For instance, we need two types of properties:

- One is to protect the IRM data, and
- The other property is to control flow from monitored code to IRM code.

So we need some kind of Control Flow Integrity such that IRM code will not be exploited by the monitored code. So some properties of control flow transfers should be enforced. We can enforce it by checking the property before making a transfer. CFI simply says to verify certain criteria regarding jump/call targets. Let us explain this with example. One of the policies for enforcement can be:

1. **Target of control-flow transfer should be within the code segment.** It protects against code injection attacks. It is important to note that while doing this we are not making many assumptions. In contrast, in the context of buffer overflow defenses, we made several assumptions, e.g., the protected program will not actively try to subvert the defense. But here we have not made such assumptions here, still it guarantees that no code injection will be possible. (Note: there are some assumptions underneath, if you looked very closely at low level details relating to machine code. But the level at which we see, we don’t need to have any assumptions to be made as long as we perform this check any time a control flow takes place.)

   Does it protect from **return to libc** attack? The answer is no. Because libc is within the code segment and the check is rather coarse saying that the jump cannot go outside the existing code.

   It is certainly a useful primitive because when you say you cannot execute injected code --- it gives you certain guarantees that you can build on. Let us talk about an example. Let’s think about address space randomization. It is reasonably secure against the remote attacks. But let’s say there is some vulnerability and there is some target that is not protected that allows someone to inject code. Notice that this not enough because of the randomization. Let’s say the attacker’s goal is to execute some system API function, say CreateProcess on Windows. Now he has to figure out where is that system function. He will not be able to directly call the intended function and execute his malicious activity as he does not know the location of this function (due to randomization). But the moment he has the injected code running, he can start scanning the memory and he will be able to find out the location of desired functions. There are lots of places he can look for, such as, there are linkage tables where these entry points for dynamically linked libraries are put, or he can simply scan the code memory, say, based on the contents of the first 16 bytes of a routine that may uniquely identify the routine.

   Thus the idea is that, though there is address space randomization, but if there is one weakness in the system that allows injected code to run, then everything is lost. Injected code can now figure out all the randomness that is used. So the point is that if you couple the address space randomization with the technique like this (enforcing checks prior to jump and limiting within code segment), the fragility of address space randomization can be mitigated.

Thus you end up putting checks like this prior to each control flow transfer. However, as discussed before, malicious code may attempt to jump past the checks. Since we are only checking that the target is within code segment, it is possible for malicious code to jump past checks. However, in principle, the control-flow integrity technique should be able to protect itself: the threat being addressed is one of jumping to a location, and hence an appropriate CFI property should be able to prevent it. Let us consider the following criteria in this regard:

2. a) **All calls should go to beginning of functions,**
   b) **All returns are to call sites known at instrumentation time**,
   c) **All jumps are to instructions in original code, except for jumps to control-flow transfer instructions --- these jumps to should instead transfer control to the checking instructions that precede the control-flow transfer.**

   Here point (a) is trivial to understand. Point (b) says that any return should go to an instruction that follow a call. You might think it is better to enforce a stronger condition that returns go back to the call site; but enforcing this requires the CFI mechanism to maintain its own stack, which increases complexity. So we relax the condition here. We are simply saying that returns can go back to *any* call site. Point (c) addresses the question of jumping past checks, or otherwise attempting to interfere with the CFI checks. Note that 2(a) (b) (c) imply rule 1, provided all the targets are based on information that is statically available. (For instance, the implication won't hold if (b) were implemented by checking at runtime if the return address was preceded by a call instruction --- in that case, malicious code could create some code on the fly in data segment, and transfer control to that code by simply putting a call instruction before the desired target location.)

Implementation of CFI:

In terms of implementation issue, there are **direct jumps** and **direct calls**. Again there are **indirect jumps** and **indirect calls**. Direct means the target is specified in the code itself. For direct jumps, the check itself can be performed at the instrumentation time. Check need not be done at runtime and this is one reason why these techniques are reasonably fast. (Most jumps/calls are direct, so they dont need runtime checks.)

All indirect control-flow transfers, including returns, need to be checked at runtime, because the target of control-flow transfer are not statically known.

There are various ways to implement CFI.

Alternative 1: One simple approach is the one that **uses some sort of global Boolean array** denoting the valid addresses initialized at the beginning of the program. Thus every jump is checked against the target, and if it is fine, then the jump is made. The global array itself needs to be protected. You can do that by making the array read only. (If it is not read only, that is, if it is writable, then malicious code can modify the global array and can jump anywhere it wants.)

Limitation: This simple alternative has two limitations. First, it needs significant amount of memory, say, 1/32th of address space (if all jump targets are required to be 4-byte aligned). Second, it assumes that the validity of the target is independent of the source address.

Alternative 2: The approach is based on the idea of pairing sources and destinations. Let's say X is a destination for a call. In this approach, they are going to write a constant, say k_x, just before X. Every time when somebody is going to call X, you will do compare (X-4) and k_x. That is, you are going to check the location 4 byte prior to your control flow transfer with k_x. This is of course makes no sense to check at runtime if you already know what the value of X at compile time or instrumentation time, then you can check it right away. But if you don't know the value of X at compile time, then the check has to be performed at run time. This is of course a very simple and efficient technique. There is no complicated data structure involved. You insert constants in the binary. As it is binary, it is already read only.

Though the mechanism is simple, but there are some difficulties in using it.

The first problem is that if I don't know X statically then how can I generate k_x. So what I will have to do is that I have to see what the set of all possible values it can have. If I can estimate that, then I am going to give the same constant value to all those functions. Let's say X can point to f, g or h functions. If I know that those are the only possible values for X, then I am going to associate the same constant with all three functions. And that is the constant that will be used in the compare. This will immediately lose some precision. This means, any where a call to f is accepted, a call g or h will also have to be accepted.

The problem can be worse. Let's say there is another place where there is an indirect call using Y, where Y can be g, i, or j. So if I have to put one constant for all three g, i, j and one constant for all three f, g, h, then that means I have to use single constants for all the five functions. That means the approach starts losing precision reasonably quickly. Then if you use the same technique for the returns, it will become even harder. So there is a precision loss though in terms of implementation it is relatively simple. Again there are also some cases you will have no idea what these function pointer values will be.