Today Discussion:

Today’s discussion involved the discussion on project related to google project “Native Client”. It means you can run native code in browser. You can run random native code that you download from internet and do not trust in the browser and How do they make it possible, is very interesting which we will discuss today.

It is basically dependent on the old idea of fault isolation.

1) General problem in security today is we often want to break up the main security system (large one) into many independent small systems. This is done for reasons: -
   a) Independent system may not really trust each other. You and your browser don’t trust the code you download of the internet. Here comes the idea of Packet filter rules. If you are running the system and you want to run the packet sniffer. It is a program that connects it to kernel and asks for packet if you see it for any process. Basically used for monitoring the network. If we use packet sniffer on really high speed network, then we can get overwhelmed with data. So the packet sniffer only wants to do is to ask the kernel, give me the packets which match the certain rules.
   b) So, Basic idea is to capture packet and transfer it to user space, kernel captures it and send it to the program and then application might throw it, it is expensive. So packet filter are program that can be downloaded into kernel space and they execute inside the kernel to filter out the packets before passing it to user space. Do we understand the problem here? We do not want arbitrary code to be running inside the kernel. It is not good for security purpose.
   c) One case might be, if you have untrusted code running inside larger program is like some database. You might be able to define function that you can query or we can use inside the query to query data into
Rather than getting whole database into app and then querying it, we can query and bring the data into application.

d) This is an example which is similar to native code where plugins that are not trustable.

e) Another problem that comes up is sometimes we want to split the programs and expects them to trust each other but in order to embrace them, we want to separate them. And then we can give each component the separate privileges. Separation of privileges.

f) So whenever we want to separate program like this, what can be done? What does google chrome do with plugins?—It’s a separate process. It is good way of doing things. In this way you are trusting the OS and hardware by doing memory isolation but downside is overhead. Context switching between them, copying data between them.

g) In general running two different distrust process in their own space is appropriate only when level of communication between them is not large, as we might have to do lot of context switching between the processes, may be thousands of thousands of context switching. So, it’s not going to be a simple workable solution.

h) Native client is an example of system that is trying to run two mutually distrust process running in their own address space and inside the same process. So you can get very light weight boundary crossing between the distrusting.

Let’s see in details about the concept.

General Techniques: -

INLINE REFERENCE MONITOR AND NATIVE CLIENT AND SOFTWARE FAULT ISOLATION AND OTHERS

- Lot of projects in this area.

If you are going to run two different things inside the same address space, and they don’t trust each other, or may be one of them distrust the other, like downloaded code trust the browser but browser doesn’t trust the downloaded code. So we can start from where one component is trusted and another one is untrusted. So if we run the code we don’t trust that is the
native code here, so in general if you going to run the block of untrusted binary code in the same process and address space as your trusted code. What kind of security problems we can face and we need to solve? : -

- **Memory isolation** - We should restrict untrusted component to only read, write its own memory. So the kind of picture in mind you can have is:

![Virtual memory space diagram]

- **Controlling privileged actions (e.g. syscalls, instructions etc.)** - There is a challenge that comes up which we will discuss later. Question raised here was “Doing it at software level as it is already there in hardware”? Isn’t it overhead? It is not much, only 5% overhead. Doing it at hardware level might take it to the level of thousands and thousands of instructions needed for context switching, so it’s going to be an interesting balance. We will be leaving the protection to hardware level and occasionally paying the way for context switch might be the efficient way to go.

- **Chroot Jails** is an operating system level feature where you essentially have a sub-tree of your process tree that has a different notion of the root directory is.

- If we perfectly isolate memory and we imagine we prevent from executing any system call, the final piece we want to put in place would be the one below:
How to make Cross-domain calls? -> how to call the trusted component from inside the untrusted component or sometimes the untrusted piece of software want to use a service provided by the trusted software and so it should be able to make a call to that service provider directly. So for example if we want the untrusted software to open a file, perhaps in the trusted software we will provide the library that’s going to be like safe open. We try make a call to safe open from untrusted program to that open in trusted program. So safe open will try do some checks. So we would want to make a restriction on the untrusted program. Example: trusted program can be our browser and native plugin into the browser is untrusted, something which is downloaded from the browser which is separate process.

The issue in stack and heap is there in the first two part of memory isolation and controlled privilege section which will be discussed later.

For memory safety-

1) We are going to insert runtime checks into the untrusted binary, so every time an untrusted binary is about to do a memory reference it will do check to see if that check is safe.

2) Preventing Bad jumps. Prevent untrusted binaries from jumping over these runtime checks.

3) Verify correct compilation. The person running the code, have to check that code is compiled properly to insert the check. Problem here is we have to trust the person who inserted the check. So the person has to verify. This can be understood with the figure below on the next page: -
Key here would Code producer will produce the code that is easy to verify and consumer will deploy a simple verifier that will check the code obey certain rules. Like doing bounds check, or before jump it does something else, so that it do not jump over the bounds check.

**A big challenge here would be, verifier would have to look at .O and so important problem to solve would be to make .O that is easy disassemblable.**

Verifier is an external step. It is a fixed step of rules that binary have to meet. We have to perform verification only once. This can be done outside the system. If it says yes, we save binary to disk. If no – we throw out of the system.

Verifier is a separate process. Once it is done, browser can load the untrusted binary into its address space and run it without any verification. Verifier will only verify low level rules which are universal. They are necessary for constraining the application so that it is enough for you to force the policy. And whatever we do after that is system dependent.

**Problems with this approach:**

If code has a check for the safety of access beforehand, evil binary can get around that check? How it will get around let’s see that.

**{Check bounds on a memory access}**

**Jump ------→ Then store %r0, [%r1]**
This can be bypassed, by doing jump. So we have to avoid it.

Let’s imagine application has a load instruction. Example we consider. We want restrict it to see that it is safe.

\[
\text{Load} \left[\%r0\right], \%r1
\]

What we should do to ensure it is safe? We would insert a check before it.

Keeping in mind there are 16 registers on x86. New rule for check would be: - \%r14- Lower bound, \%r15 upper bound. So whenever control is transferred from trusted to untrusted program, it sets up \%r14 as lower bound and \%r15 as upper bound.

![Virtual address space diagram](image)

So, we can make a branch instruction B1 before the instruction:-

\[
\begin{align*}
\text{B1} & \ \%r0, \%r14, \text{.ERROR} \\
\text{B1} & \ \%r15, \%r0, \text{.ERROR} \\
\text{Load} & \left[\%r0\right], \%r1
\end{align*}
\]

So, we can optimize this by one way. Let’s suppose we do the same thing as baggy bounds to optimize it taking it powers of 2. So we made the region aligned to the multiple of two and they were of size of same power of 2.
Base of memory region can be found out with the help of PC without extra thing. We can consider taking program counter and we can do something like this below:

- \texttt{shl} - shift left
  \[
  \text{shr} \ %PC , \ k , \ %r2 \\
  \text{shl} \ %r2, \ k , \ %r2 \\
  \text{shl} \ %r0, \ 64-k , \ %r0 \\
  \text{shr} \ %r0, \ 64-k , \ %r0 \\
  \text{Or} \ %r2, \ %r0, \ %r0
  \]

This can be seen with the help of picture below about what this operation did.

\* So, here we can ensure that as long as they execute this, it wont point outside the trusted code.
Problem to solve here is how to prevent this instruction to jump over? **Native client does the tricks of brute force.**

➜ There is two kinds of jump that occur in a program.

- **Absolute jump, jump to 0x12345678.** Do we have to care about that jump? **No, verifier can check it in advance.** If we solve disassembler problem then we can check it in advance that whether this address is in bounds or not.

- **Jump %PC + OFFSET which is constant** -> if we completely disassemble the full code, **we can check where it jumps to.** Verifier can check where it falls inside the code. It is jumping to specific instruction so verifier can check it.

- The hard part is when we have jump like **jump %r0** (god only knows where). **This cannot be statically verified.** It is the function pointer or if you have a large switch table which specify to which block it jumps to. So it is a pointer to object itself not a function pointer. We have to check **whether it is inbound. Jumping into untrusted code.** We can do the same thing of bound check by doing this: -

  1) **Bounds check %r0** (straight forward)
     - How native clients solve it? -- > Check that %r0 = 0 mod 32.
     - Native clients generate code inside the untrusted code. Leys see figure.
Question here comes, how are we going to handle ret instructions?

- What is a ret instruction? It is like pop and a jump. And so if we can make sure that pop is safe and then we can check that stack pointer points in range then it cannot read memory out of the range.

So let’s list the set of rules that verifier uses and how we going to solve disassembly problem - :

- All jumps 0 mod 32
- All jumps bound check
- All loads/state bounds check
- No instruction crosses 32-byte boundary
- Checks must be in the same block as the instruction they guard.

Disassembly problem: - Biggest challenge with x86 instruction set is that we do not know where the instruction began and ends. But if code is compiled with this rule, that problem goes away. Because all we have to do is start disassembling the first block and you verify that within this block all the memory addresses are bounds check and all of the jumps are bounds check and checks of 0 mod 32 and all the steps as mentioned above.

What does compiler do if it is generating code and it gets to a point where it has to puts several bytes instruction here and instructions is just blocks? Insert no ops for padding. No-ops on x86 are single bytes. No matter how much padding space we need to put in, it’s a single byte which we can always fill in.

Flow based disassembler?

- It’s a recursive disassembler, when you present arbitrary program to it, it begins parsing instruction till it encounters jump, go to where jump will jump and start parsing instructions there and then start to disassemble the target of branch and instruction following the branch and so on and eventually disassemble everything.
- Problems with that is, you can trick it into disassembling extra stuff which it is not supposed to do.
- Suppose you have a branch in your program where you have coded it, it will always be true, but disassembler may not be able to figure it out statically because the way we compute the branch is really really
complicated, because we know that it comes out to be true and so disassembler assumes that it can go either way so we can trick the disassembler in disassembling the bunch of code which might be garbage.

**Linear sweep disassembler**

- It is way to go for native client. We can take all the code break into 32 byte chunk and disassemble each chunk independently from one another. Rule here is no instruction crosses the 32-byte boundary. Every 32 byte block can be disassembled independently of another one.
- **Basic block is sequence of instruction which do not have jump but ends with a jump.** It might be bunch of code having ads and multiplier. Here 32 bytes of block do not belong to basic blocks of program. They are just 32 bytes disassembling blocks.
- **0 mode 32 is a simple trick but it so easily solve the problem.**

How do we prevent sys call?

→ Extra check to check no sys call. If I see that instruction in disassembler we reject it.

Class Concluded