Native Client:
How native client works?
We can run native code in browser and more importantly we can run the random native code from the Internet that we do not trust in the browser. And this can be done by implementing an older idea, so it is a revolution of new idea called software fault isolation.

A general problem in comp security is that we often break up a large system into many small independent systems. The independent system does not trust each other. For example: the code downloaded and the browsers do not trust each other.

Concept of BSD, packet filter, net filter
So if we are running a system and use a packet snippet. A packet snippet, it is a program that connects with the kernel and says to the kernel to give all the packets at any time. So we run it on a high speed of network then we will get overwhelm of data. And so a common thing that packet snippet does is that it ask the kernel to give the packet that acts on certain rule, like the packet, which has length greater 14,00. So overhead is to capturing the packet and passing to the user space. So packet filter are program, which can be downloaded in kernel and the kernel executes the program on the behalf of the user in the kernel space before it get passed to user space. So the problem we can’t allow any arbitrary code to run in the kernel space because of security.
The other case is that when we have an untrusted code running inside a larger program like in some database, I can define a function, which can query or we can use inside the function to query data into. Rather than getting whole database into app and then querying it, we can query and bring the data into application. So if the database is used by multiple users and they upload random functions into it. Then we have to make sure that the database remains secure. This is an example, which is similar to native code where plugins those are not trustable.

Another thing that comes up that we want to split the program that trust each other in order to embrace them, we want to separate them. Then we can get each component less authority. For example if we have a program that read and writes stuff to the network. We can break into two programs, one, which can read from the network and can communicate locally and other half write to the network. So suppose we find a bug in read, so kernel does not allow the program to read but write to the network and vice versa. So this is a general principle called separation of privileges.

So, whenever we want to separate the program like this we can do so by making separate piece of code and make them isolated.

What does Google Chrome do with plugins?
It runs the plugins with separate processes. It is a good thing, as we trust the OS.

What is the downside of running two things in separate process?
We have overhead, context switching as you have to copy data between them. If suppose we have to make 100,000 system calls between different process. So separating the process will not be a good solution.
So native client is a system, which puts two different processes in same process and same address space. So we get very light boundary crossing.
So lets talk about how we are going to do this:

**Inline Reference Monitors & Native Client (& Software Fault Isolation & others)**

So if we have to run two different things in same address space and they do not trust each other. For example, the downloaded code trusts the browser but the browser does not trust the downloaded code. So we have an untrusted component and a trusted component. So we have to run code here that we do not trust. We are talking native code here. So, if we are running an untrusted code in the trusted code then what are the problems associated with it:

1. **Memory Isolation**
   So we should restrict untrusted component to only read/write its own memory.

<table>
<thead>
<tr>
<th>Trusted Region</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Untrusted Region</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Trusted Region</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Virtual Memory Space</td>
</tr>
</tbody>
</table>

So we have to isolate the untrusted and trusted code.
2. **Controlling privileged actions** (e.g. syscalls, instructions, etc.):
If we have to adapt the system in which we have to load untrusted code into the kernel, there might be instructions that we do not want to execute. We do not want the untrusted code to open file in trusted code. There are two things privilege and unprivileged mode. Suppose the browser is running in privilege mode and browser’s plugins are running in unprivileged mode. So if browser’s plugin wants to modify something then it will fail.
3. Cross-domain calls:
If we isolate the memory that untrusted program is trying to access and prevent it from execute or any system call then how the untrusted component calls the trusted component. Suppose the untrusted software is software provided by the trusted software. It needs to make call out, for example we do not want the untrusted component to open files itself and perhaps in the trusted soft we provided a library that allows the untrusted to call.

Example of trusted and untrusted component:
Trusted program might be the browser. In native client, there is a browser in one process and a native client subsystem plugin to the browser, so that will be a separate process. If we visit a web page that has a native client applet or a piece of code that will be untrusted program.
The trusted and untrusted are re-locatable code.

Solution to the problem:
The way we are going to solve this problem is for example memory safety, we will insert checks into the untrusted binary. Every time untrusted binary is going to do a memory reference, before that it will check that whether that reference is safe.

Problem 1:
If the code has a check for the safety of the access, then we have to think of a way that evil binary get around with that check.

Step 1: Insert runtime checks into untrusted binary

```
Check bounds
On memory access
Store %ro,[%r1]
```

If this code is in the binary then we think it can get over the check. Yes, by using JUMP it gets over the check.

```
Check bounds
On memory access
Jump  ->  Store %ro,[%r1]
```

Step 2: Preventing bad jumps.
So in this we have to prevent the untrusted binary from jumping over these runtime checks.
This is what compiler is doing.

Step 3: Verifying correct compilation
If we put all the checks then the person running the code needs to make sure that the code is compiled correctly.
Diagram:

Code producer

Code producer

.C

NaCl compiler

.O

.O

OK/FAIL

NaCl Verifier

Code Consumer

Explanation:
The program has a .c code that goes into NaCl compiler and that produces a .o file. This whole happens at code producer side. The code producer ships the .o to the code consumer and the code consumer before running the code run through the NaCl verifier. If the NaCl says that this code does not verify it then it will not run it. But if it says its ok then this .o will run inside the system.

So, the important thing would be the compiler is designed produce code that is easy to verify. So finally we have a simple verifier that checks the code based on certain rules. One of the rules that it is going to check is that it should not jump over the bounds checking.

The big challenge here is to make .o, which will be easily dis-assembled. The verifier has a set of rules that binary code has to meet. In fact we have to perform the verification once.
Next Topic:
If we are going to limit the memory that the app can load then let’s think about that the application will have a load instruction and we have to restrict it so as to make it safe.

Load [%ro], %r1

So up here we want to do a check. So we have lot of registers floating around. Suppose, there are 16 register registers in 0x86 architecture.

New rule:

%r14 – lower bound
%r15 – upper bound

So before the code transfer control from one point to another. It sets %r14 to point below the untrusted region and %r15 to point to above the trusted region (see the diagram below).

Making branch instruction:

So,
b1 %r0, %r14, .ERR (these are jmp instruction)
b1 %r15, %r0 .ERR

So we have a couple of instruction. This is a simple method. But it runs a couple of registers. And therefore it is not that great as it consumes lot of instructions. So we have to optimize this.
How we are going to improve this?
Like we did in baggy bound checking. We have to make all regions of same size and with power of 2. So we have to do the same thing here. We are going to assume that the code is inside the untrusted region in order to save the registers. So if we want to figure out the base of this region then we can do that by program counter.

The app knows the value of ‘k’.

Shr %pc, k, %r2
        -> Means that shift pc by k into register r2
Sh1 %pc, k, %r2
        -> Means that shift pc by k into register r2
Sh1 %r0, 64 – k, %r0
        -> Means that shift r0 by (64 – k) into register r0
Shr %r0, 64 – k, %r0
        -> Means that shift r0 by (64 – k) into register r0
or %r2, %r0 %r0

k bytes

<table>
<thead>
<tr>
<th>%pc</th>
<th>PC_k</th>
<th>PC_k-1</th>
<th>...</th>
<th>PC_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r0</td>
<td>r_0</td>
<td>r_1</td>
<td>...</td>
<td>r_k</td>
</tr>
<tr>
<td></td>
<td>PC_k</td>
<td>PC_k-1</td>
<td>...</td>
<td>PC_0</td>
</tr>
<tr>
<td></td>
<td>r_k</td>
<td>r_k-1</td>
<td>...</td>
<td>r_0</td>
</tr>
</tbody>
</table>
The above commands are explained in the diagram above. What it does is that it takes %pc and %ro, which have the values as shown in the diagram, and produces an output shown below the line. So we finally modified %ro. So by writing some instruction before ‘Load’, we have made sure that it will not access the memory outside the untrusted region.

Another Problem: How to prevent code from jumping over these instructions

There are two kinds of jumps.

1. Absolute jump: that is jump to any location given. We just need to check that this address is in bounds or out of bounds.
   
   \[0x12345678\]

2. Second jump is
   
   \[\text{jmp } \%\text{pc} + \text{OFFSET}\]

   this is constant

It we completely disassemble the code then the verifier can check where it jumps. It effectively jumps to specific instruction and it will check that whether it jumps in the middle of these things. The harder part is that if we jump like shown below:

\[\text{Jmp } \%\text{ro}\]

This we cannot statically verify. Sometimes if we have larger switch table. It comes up in real code. It is pointer to the object itself has a pointer to the V table. So we have to make sure that it should be in bounds and make sure that it only jumps in other untrusted code. For this we can,

1. boundscheck %ro
2. check that %ro = 0 mode 32.

The native client generates code inside the untrusted code. Suppose the region shown above is all untrusted code. It will divide the region into 32 byte chunks. There is a rule wherever there is a load and load is preceded by three instructions. Then the load along with 3 instructions should fit inside same 32 byte block. So we can have like,
The initial load instruction is with or, and, and. So the jmp operation can be performed in the same block of 32 bytes and not outside. So before executing the jmp instruction it checks for 0 mode 32.

**Assembly Problem:**
In ROP that we have talked about earlier. One of the tricks of ROP is that we have a dense instruction of x86. So there are challenges of dis-assembling x86 code. But if this code is compiled according to the rule the problem goes away as all we have to do is to start disassembling the first block, and verify that in this block all memory bounds check. No instruction can cross 32 bytes block boundary so if we have 5 byte instruction then it would be illegal to cross this boundary.

**Verifier:**
1. All jumps 0 mod 32.
2. All jumps bounds checked.
3. All loads/staces bounds check
4. No instruction crosses 32-byte boundary
5. Checks must be in same block as instruction they guard.

So the compiler has to do if it is generating code, and if it has to put 7 bytes instructions and this instruction do not fit into the block. So the compiler will display a no ops error with this problem. No ops on 0x86 are single byte.

Problem is given below:

![Recursive disassembler](Image)

**Recursive disassembler:**
When we are presented with an arbitrary code and we do not know what it does, so it parses instruction till it hits the jmp, and goes where jmp will jump and then parses
instructions there. Then it starts disassembling a part of the branch and finally disassembles everything.

Problem with recursive disassembler:
The dis-assembler does not know how big the array is. The missing code is really bad thing here.

**Linear disassembler:**
We can take all the code and break it into 32 byte chunks. No instruction crosses 32 byte boundary.

**Bug in verifier:**
It cannot escape the code in untrusted memory segment and also cannot escape the untrusted code segment.

**Preventing system calls:**
So there is an instruction to prevent the system call in the dis assembly. If we see that instruction then there will be no system calls.

Class Concluded