Shadow Stack

Keep a copy of the stack in memory

- On function call: push the return address (EIP) to the shadow stack.
- On function return: check that top of the shadow stack is equal to the return address (EIP) on the stack.
- If there is difference, then attack happens and the program will be terminated.

Shadow stack requires the support of hardware

- Intel CET (Control-flow Enforcement Technology):
  - New register SSP: Shadow Stack Pointer
  - Shadow stack pages marked by a new "shadow stack" attribute:
  - only "call" and "ret" can read/write these pages

StackShield

A GNU C compiler extension that protects the return address.

Separate control (return address) from the data.

- On function call: copies away the return address (EIP) to a non-overflowable area.
- On function return: the return address is restored.
- Even if the return address on the stack is altered, it has no effect since the original return address will be copied back before the returned address is used to jump back.

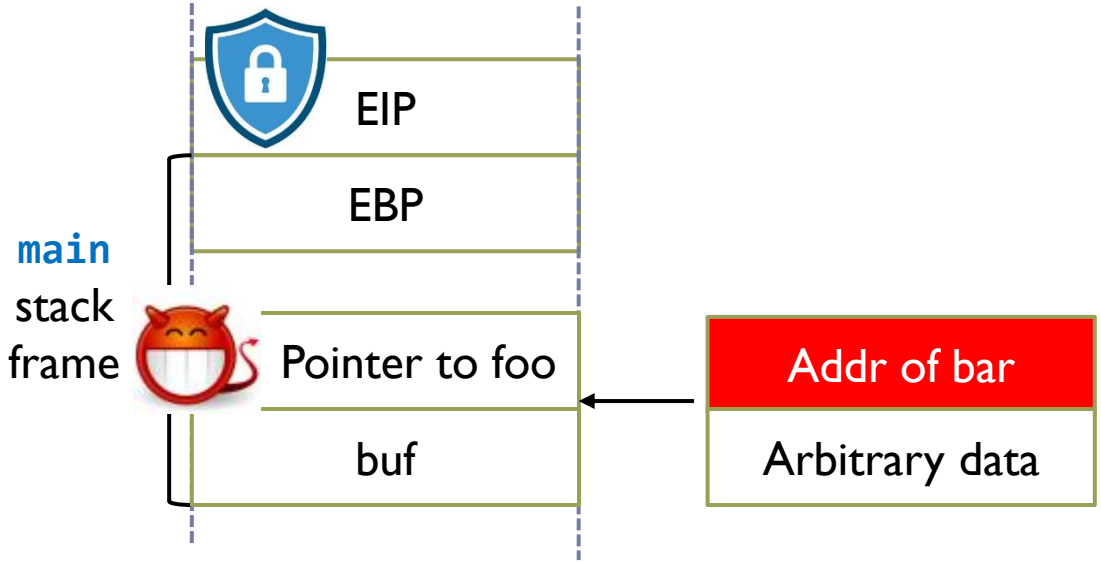
Common Limitations of StackGuard, Shadow Stack, and StackShield

Only protect the return address, but not other important pointers

Hijacking a function pointer

- Even if the attacker cannot overwrite the return address due to the canary, he can overwrite a function pointer.

```
void foo () {...}
void bar () {...}
int main() {
    void (*f) () = &foo;
    char buf [16];
    gets(buf);
    f();
}
```



PointGuard

A compiler-based approach to protect the function pointers from being overwritten

- Encrypt all points while stored in memory, and decrypt them when loaded into CPU registers for use.

Steps

- A secret key is randomly generated for each program when it is launched.
- Pointer encryption: when loading a pointer to memory, encrypt it with the key (typically XOR).
- Pointer decryption: before a pointer is used by CPU, decrypt it with the key (XOR). This ensures the pointer is in its original, unencrypted form only during actual use within CPU, minimizing the window of vulnerability.
- Without knowing the correct key, the attacker cannot overwrite stack data with the encrypted form of malicious function address.

Pointer Authentication

Introduced in ARM architecture to protect function pointers

- Appending a cryptographic signature, known as a Pointer Authentication Code (PAC) to pointers.
- Allowing the CPU to verify the integrity of pointers before they are used.

Steps

- Pointer signing: when a pointer is created or updated, a PAC is generated using a cryptographic hash of the pointer's value and a secret key. This PAC is then embedded into the unused high-order bits of the pointer.
- Pointer verification: before a pointer is used by CPU, the system verifies its integrity by recalculating the PAC and comparing it to the one stored in the pointer. The pointer can be used only when the PAC values match.
- Without knowing the correct key, the attacker cannot generate the correct PAC for his malicious function pointer, and pass the pointer verification.

Recall: Steps of Stack Smashing Attack

1. Find a buffer overflow vulnerability in the program
2. Inject shellcode into a known memory address
3. Exploit the buffer overflow vulnerability to overwrite EIP with the shellcode address.
4. Return from the vulnerable function.
5. Start to execute the shellcode.

Solution:

- Non-Executable Memory

Non-Executable Memory

Key idea

- Attackers inject the malicious code into the memory, and then jump to it.
- We can configure the writable memory region to be non-executable, and thus preventing the malicious code from being executed.
- Windows: Data Execution Prevention (DEP)
- Linux: ExecShield

```
# sysctl -w kernel.exec-shield=1   // Enable ExecShield
# sysctl -w kernel.exec-shield=0   // Disable ExecShield
```

Hardware support

- AMD64 (**NX-bit**), Intel x86 (**XD-bit**), ARM (**XN-bit**)
- Each Page Table Entry (PTE) has an attribute to control if the page is executable