Article

Improving control flow integrity with pointer authentication

Increase confidence that your code uses pointers correctly.

Overview

Many common attacks against software use techniques that compromise control flow within an app, executing code within the app that the attacker wants to run instead of the intended code path. Two examples of such attacks are:

Return-oriented programming (ROP)
: The attacker manipulates the call stack to cause functions to return to the wrong locations.

Jump-oriented programming (JOP)
: The attacker manipulates the heap to cause an indirect jump instruction to jump to the wrong location.

Pointer authentication provides probabilistic protection of control flow integrity (CFI) by annotating pointers with signatures. You sign a raw pointer to produce a signed pointer, which contains an embedded pointer authentication code (PAC). To use a signed pointer, you authenticate it, which validates the PAC and, if valid, returns the raw pointer. Otherwise, if the authentication operation detects that the pointer is invalid, it returns a value that represents an invalid raw pointer. Your process generates a segmentation fault and a crash report when it uses the invalid raw pointer. For more information, see “Recognize pointer authentication failures” in Preparing your app to work with pointer authentication.

> Note
>
> The authentication operation might fail to detect that a value is invalid, and return a usable value even though its input is invalid. The authentication check is a probabilistic signal of confidence, not a guarantee that the pointer isn't compromised.

To generate a PAC that you use to validate a pointer, you need to identify the *signing schema* that the system uses to sign the pointer, which is made up of the signing key and a discriminator.

You identify a signing key by name; the CPU keeps the key's data secret and can use different values for the same key name in different processes. Compute discriminator values as needed. For more information, see the section "Generate a discriminator", below.

- In most situations, for example, storing a function pointer in a data structure, you use the `__ptrauth` type qualifier on the field where you store the pointer. This tells the C compiler to store the pointer more securely than it otherwise does by default. Using the `__ptrauth` type qualifier supports several simple, but effective, pointer authentication schemas.
- In other situations, for example if you're writing a just-in-time (JIT) compiler and you protect a pointer to a function that your JIT compiler creates, you might need to use a compiler intrinsic to sign the pointer before you pass it around in yout code. Additionally, use compiler intrinsics to gain more control over the signing schema, taking care to ensure that an attacker can't alter the discriminator in the same way that they can alter a signed pointer.

The pointer-signing key names intrinsic operations are defined in the header file `<ptrauth.h>`.

Generate a discriminator

Signing pointers requires an arbitrary value called the *discriminator*, that the processor uses as a salt for the signing operation to ensure that signed pointers for different purposes aren't interchangeable. Using different discriminators in different places makes it more difficult for an attacker to compromise your code by replacing a valid signed pointer value with a different valid signed pointer.

A discriminator is an arbitrary 64-bit value. Typically, you use either a constant value, or a value derived from the address in memory where the pointer is stored. Both of these approaches are supported by the `__ptrauth` type qualifier.

You can choose a constant discriminator value directly, or generate one based on a string using `ptrauth_string_discriminator`:

```
const int discriminator = ptrauth_string_discriminator("My discriminator string");
```

Alternatively, combine a constant value with the pointer's location in memory using `ptrauth_blend_discriminator`:

```
#define SEED_VALUE 0x1234567890abcdef

const int discriminator = ptrauth_blend_discriminator(pointer, SEED_VALUE);
```

When you use a discriminator that incorporates the pointer's memory location, you can't copy the pointer or a data structure that contains the pointer using `memcpy` and related functions because the discriminator isn't valid to verify the pointer at the new memory location.

Select a key

The cryptographic key that the system uses to generate a PAC for a pointer is known as the *signing key*. You identify which signing key to use by name, and the system doesn't give you access to the signing key's value. Choose from the four different signing keys, depending on whether you're protecting a code pointer and whether you need a process-dependent or process-independent signing key:

`ptrauth_key_process_independent_code`
    A process-independent key you use to sign code pointers.

`ptrauth_key_process_dependent_code`
    A process-dependent key you use to sign code pointers.

`ptrauth_key_process_independent_data`
    A process-independent key you use to sign data pointers.

`ptrauth_key_process_dependent_data`
    A process-dependent key you use to sign data pointers.

Signing a pointer with a code-signing key produces a larger PAC than a data-signing key, which increases the protection of pointer authentication. In each case, the size of a signed pointer is the same as the size of an unsigned pointer, and the PAC is stored in unused bits of the pointer.

In most situations, use a process-independent signing key. The system uses process-dependent keys to protect particularly high-value pointers such as return addresses and frame pointers, and if you use the same keys for other contexts you increase the risk that your app re-uses the same signing schema for these pointers.

The header file `ptrauth.h` also provides these names that are synonyms for the basic signing keys, that you can use to provide extra information in your code about a protected pointer's purpose:

`ptrauth_key_function_pointer`

A key you use to sign function pointers.

`ptrauth_key_return_address`
A key you use to sign return addresses on the stack.

`ptrauth_key_frame_pointer`
A key you use to sign frame pointers on the stack.

`ptrauth_key_block_function`
A key you use to sign pointers to block functions.

`ptrauth_key_cxx_vtable_pointer`
A key you use to sign C++ v-table entries.

Annotate pointers with the pointer-authentication type qualifiers

Use the `__ptrauth` type qualifier to tell the compiler to generate a PAC for your data pointer or function pointer, and to validate the PAC when you dereference the pointer. The type qualifier takes three arguments:

`key`
A constant expression that identifies the name of the abstract signing key to use, discussed in the “Select a key” section above.

`address`
A Boolean that indicates whether the compiler needs to use `ptrauth_blend_discriminator` to vary the discriminator based on the pointer’s address.

`discriminator`
A constant expression that the system uses as a salt in generating the PAC.

For example, to declare a data pointer that the system signs using the constant discriminator value `0x1f35`:

```
void *__ptrauth(ptrauth_key_process_dependent_data, 0, 0x1f35) *pointer = &data;
```

Sign a pointer

To sign a pointer, you need three pieces of information:

- The raw pointer. While you can sign the NULL pointer, the resulting signed pointer has a non-zero value, and code that tests for NULL by comparing the pointer’s value to `0` gets the wrong result. Therefore, you need to test raw pointers for NULL and sign non-NULL raw pointers.

- The abstract signing key, discussed in the section “Select a key”, above.
- A *discriminator*, an arbitrary value discussed in the “Generate a discriminator” section above.

Important

Don’t store the discriminator alongside a signed pointer in memory if you generate different discriminators for different categories of pointers in your app. Doing so makes it easy for an attacker to replace both the signed pointer and the discriminator that the system uses to validate the PAC.

To produce a signed pointer for a constant address, in this example a data pointer:

```
void *signed_pointer = ptrauth_sign_constant(pointer, ptrauth_key_process_dependent_
```

The returned signed pointer is the same size and type as a raw pointer, and the signature information is stored in unused bits of the pointer value.

Authenticate a signed pointer

To recover the raw pointer value, if the signed pointer passes validity checks, call `ptr_auth_data`:

```
void *raw_pointer = ptrauth_auth_data(signed_pointer, ptrauth_key_process_dependent_
```

To recover a pointer value that’s signed for use as a function pointer, if the signed pointer passes validity checks, call `ptrauth_auth_function`:

```
void *function_pointer = ptrauth_auth_fuction(signed_pointer, ptrauth_key_process_de
```

To check the validity of a signed pointer and obtain a version that’s signed with a different key and discriminator, call `ptrauth_auth_and_resign`:

```
void *other_signed_pointer = ptrauth_auth_and_resign(signed_pointer, original_key, 
```