File: **title**

Decoding Compiler Name Mangling

cackalackycon 2025

jacob latonis

File: **slide0**

about me

staff software engineer
threat research

washington, dc

i love public transit

File: **slide1**

compilers

- what do they do
- how do they resolve symbols
- should i be scared of them

File: **slide2**

compilers (cont'd)

- lots of things
- you'll see
- maybe, probably not

File: **abi**

cpp abi

low-level Rules

- defines how compiled C++ code interacts at the binary level (memory layout, function calls, etc.)

enables interoperability

- allows independently compiled parts (even from different compilers) to link and run together

File: **slide3**

what is name mangling?

- compilers "encode" function and variable names
- avoids naming collisions:
  - overloading
  - namespaces
- essential for linker to find correct symbols

File: **slide3.1**

overloading in java

```java
class Example {
  float add(float a, float b) {
    return a + b;
  }

  int add(int a, int b) {
    return a + b;
  }
}
```

File: **slide4**

setting the stage

cpp

```cpp
int add(int a, int b) { ... }

void foo() {}

namespace abc { void foo(...) {} }
```

File: **slide5**

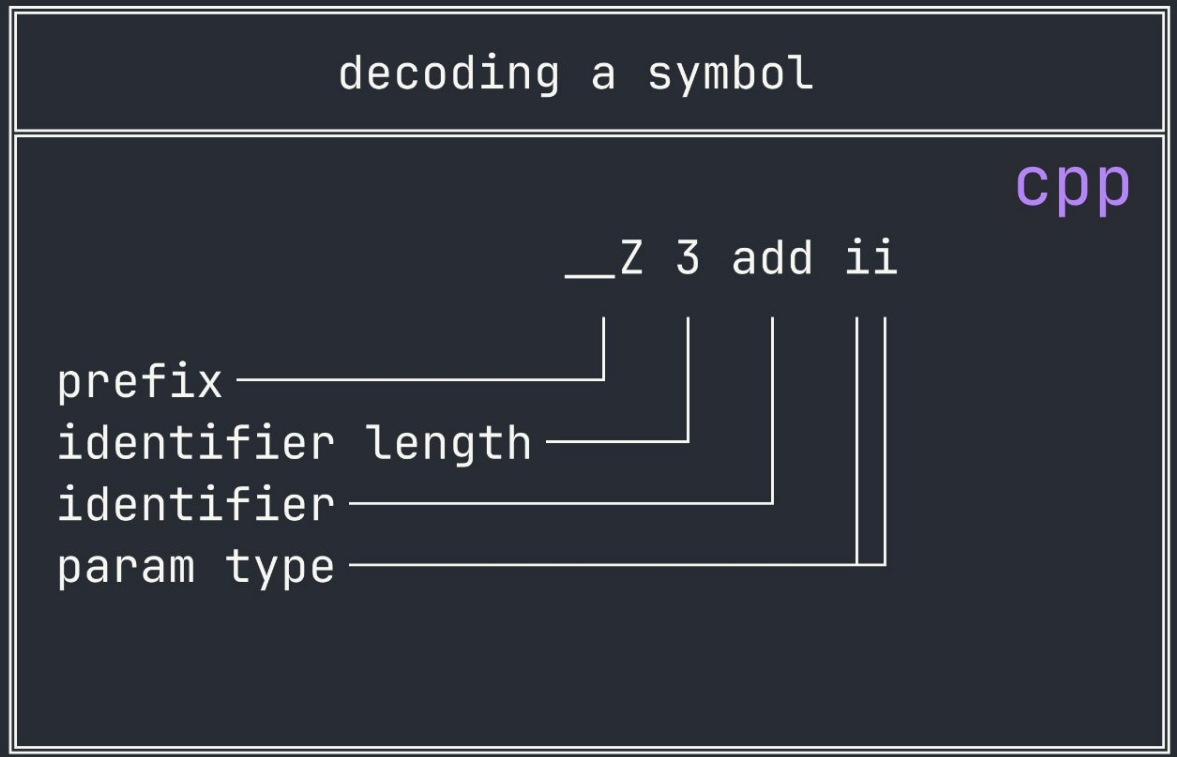
looking at symbols

▻ Command: `objdump -d a.out | grep ">:"`

Output (excerpt):

```
0000... <__Z3addii>:
0000... <__Z3foov>:
0000... <__ZN3abc3fooEii>:
0000... <_main>:
```

File: **slide6**


decoding a symbol
cpp
__Z 3 add ii
prefix
identifier length
identifier
param type

File: **slide7**

a more complicated symbol

Special chars for types/qualifiers:

- r ▹ restrict
- V ▹ volatile
- K ▹ const
- R ▹ reference (&)
- O ▹ reference of a reference (&&)
- Ss ▹ std::string (sort of)

cpp

Example:

- __ZN3dir18folderExistsAtPathERKSs

File: **slide8**

compiler variations in C++

cpp

Same function: `int add(int, int)`

clang / gcc: `__Z3addii`

MSVC: `?add@@YAHHH@Z`

No single standard for all compilers

File: **slide9**

demangling for debugging

- Debugger output often shows mangled symbols

- Understanding mangling helps you:
  - Identify the function being called
  - Recognize parameter types.
  - Navigate call stacks more easily

- Tools exist, like `c++filt` to help demangle those symbols

File: **slide10**

debugging mangled names

```
call stack example:
#0  0x... in __ZN3abc3fooEii (...)
#1  0x... in __Z3foov ()
#2  0x... in _main ()

demangled:
#0  0x... in abc::foo(int, int) (...)
#1  0x... in foo()
#2  0x... in main()
```

File: **slide11**

demangling for threat research

- Reverse engineering: analyzing compiled binaries
- Mangled names can provide insight into:
  - Functionality of code blocks
  - Libraries and dependencies

File: **slide12**

threat research (cont'd)

- demangling helps in understanding a large garbled mess produced by the compiler
- demangling may give insights into what the developer is thinking/planning

File: **slide13**

threat research (cont'd)

Imagine you're looking at a disassembler and you see this function:

```
__ZN6stream8openFileERKNSt3__112basic_stringIcNS0_11char_traitsIcEENS0_9allocatorIcEEEE
```

cpp

File: **slide14**

threat research (cont'd)

all that garbled mess really just means:

```cpp
stream::openFile(const std::string&)
```

File: **slide15**

fuzzing

targeted fuzzing
- focus on key functions.

input understanding
- know argument types.

coverage analysis
- see which functions are hit.

File: **no_mangle**

sometimes we don't want to mangle

- foreign function interfaces
  - use across languages or shared libraries

```cpp
extern "C" {
  int add_but_no_mess(int a, int b) {
    return a + b;
  }
}
```

File: **no_mangle_2**

sometimes we don't want to mangle

```
objdump -d a.out | grep ">:"

0000000100003efc <__Z3addii>:
0000000100003f30 <_add_but_no_mess>:
0000000100003f50 <_main>:
```

File: **swift**

swift argument labels

swift goes another layer deeper with name mangling

- argument labels
- parameter names

```swift
let before = s.index(before: needle)
let after = s.index(after: needle)
```

File: **swift_symbols**

swift symbols

```
objdump -d --macho example | grep index


100003e44: [...]
_$sSS5index6beforeSS5IndexVAD_tF

100003e68: [...]
_$sSS5index5afterSS5IndexVAD_tF
```

swift

File: **demangle**

i wrote a baby demangler

Find it on GitHub: latonis/demangle

- written in swift
- currently supports a subset of cpp
- plans to do swift and rust as well

- its definitely not stable yet

File: **takeaways**

takeaways

demangling reveals function signatures and class structures, aiding in understanding program behavior

name mangling can influence a lot of different operations:

- fuzzing
- debugging
- binary analysis
- exploit development

File: **sources**

wow i really want to know more
https://itanium-cxx-abi.github.io/cxx-abi/abi.html https://doc.rust-lang.org/rustc/symbol-mangling/index.html https://github.com/swiftlang/swift/blob/main/docs/ABI/Mangling.rst

File: me

where to find me
GitHub ▹ latonis Bluesky ▹ jacoblatonis.me Keybase ▹ jacoblatonis

File: **questions**