Cutting Edge Toolchain
GCC/GLiBC latest features
Linux* SW Eng. Victor Rodriguez
OSSN + ELC
21st August 2019 - San Diego CA

*Other names and brands may be claimed as the property of others
Did you know that on a Linux* system only the kernel and browser will have more lines of code than GNU Compiler Collection project?¹

LOC Linux Kernel = ~ 25 M
LOC GCC = ~ 15 M


*Other names and brands may be claimed as the property of others
“In software, a toolchain is a set of programming tools that is used to perform a complex software development task or to create a software product”
• Debugger: GDB / LLDB
• Linker: GNU ld / lld
• Assembler/disassembler: GNU gas
• Binary utilities: GNU binutils
• Emulation library: libgcc
• Standard C library: glibc

What have we build with these tools?

- Operating system kernel
- Image processing libraries
- Web server-side scripting languages
- More ..
Security
fcf-protection

There are many kinds of security attacks, one of them are:

- Return Oriented Programming (ROP)
- call/jmp-oriented programming (COP/JOP).

GCC 8 helps to prevent them.
ROP attack

Let's imagine this simple code:

```c
#include <stdio.h>

void secretFunction()
{
    printf("Congratulations!\n");
    printf("You have entered in the secret function!\n");
}

void echo()
{
    char buffer[20];
    printf("Enter some text:\n");
    scanf("%s", buffer);
    printf("You entered: %s\n", buffer);
}

int main()
{
    echo();

    return 0;
}
```

This is the hole in security.
The first 28+4=32 bytes would be any random characters and the next 4 bytes will be the address of the secretFunction.

The address of the secretFunction (0x080484b2)

```
python2 -c 'import struct; print '"'*32+struct.pack("i",0x080484b2)' | ./vuln
vrodri3@pnpxeon-server-1 /tmp $ python2 -c 'print '"'*32 + '"\x02\x84\x04\x08"' | ./vuln
Enter some text:
You entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Congratulations!
You have entered in the secret function!
Segmentation fault (core dumped)
```
We can also use Gadgets

A small group of instructions ending with a x86 RET instruction.

For example:

```
    mov eax, 10 ; ret
```

Examples for study cases:

Shellcodes database for study cases

**Description**

Although these kinds of shellcode presented on this page are rarely used for real exploitations, this page lists some of them for study cases and proposes an API to search specific ones. Thanks all for your contributions of this database but we stopped to accept shellcodes. To learn modern exploitation, checkout how to the Return Oriented Programming works.
GCC 8 introduces a new option

```
-fcf-protection=[full | branch | return | none]
```

The new `fcf-protection` option enables support for the Control-Flow Enforcement Technology (CET) feature in future Intel CPUs.
How does it works?

- The fcf-protection option checks for valid target addresses of control-flow transfer instructions (such as indirect function call, function return, and indirect jump).
- This prevents diverting the flow of control to an unexpected target.
-f cf-protection=[full | branch | return | none]

- The value branch tells the compiler to implement checking of validity of control-flow transfer at the point of indirect branch instructions, i.e. call/jmp instructions.
- The value return implements checking of validity at the point of returning from a function.
- The value full is an alias for specifying both branch and return.
- The value none turns off instrumentation.
Control-flow Enforcement Technology (CET) provides the following capabilities to defend against ROP/JOP style control-flow subversion attacks:

**Shadow Stack**
- The idea is that there is a second stack that is used only by the call and return instructions (and interrupt processing).
- When a call instruction is executed the return address is written to both stacks.
- When a return instruction is executed both stacks are popped, the addresses compared and an exception raised if they differ.

**Indirect branch tracking (IBT)**
- The ENDBRANCH is a new instruction that is used to mark valid jump target addresses of indirect calls and jumps in the program. (NOP if CPU does not support the IBT technology)

Performance
The **floop-interchange** flag applies a classical loop nest optimization and is enabled by default at -O3 optimization level and above

Let's take the next basic example:

```c
int k[1000, 100];

for (int y = 0; y < 100; y++)
    for (int x = 0; x < 1000; x++)
        k[x,y]=x*y;
```
**What we want in SW:**

```c
for (int y = 0; y < 100; y++)
    for (int x = 0; x < 1000; x++)
```

**What HW needs to do:**

![Diagram of CPU, CACHE, and Memory with Cache allocations]

<table>
<thead>
<tr>
<th>Cache 1</th>
<th>Cache 2</th>
<th>Cache 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>k[0,0]</td>
<td>k[0,1]</td>
<td>k[0,2]</td>
</tr>
<tr>
<td>k[0,3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>k[1,0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>k[2,0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>k[0,99]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>k[0,999]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Memory facts

• When the processor accesses an array element for the first time, it retrieves an entire cached line of data from main memory to the cache memory.
• If the rest of the data will be used soon, this is a major performance boost.
• If on the other hand, the rest of the data is not used, this is a net performance loss
When the floop-interchange flag is used, this code is transformed to:

```
for (int x = 0; x < 1000; x++)
for (int y = 0; y < 100; y++)
```

The memory controller is optimized for consecutive memory locations. In this scenario, the transformed code accesses memory consecutively instead of reading widely differing locations.
Code hoisting optimization

```c
int test (int a, int b, int c, int g) {
    int d, e;
    if (a)
        d = b * c;
    else
        d = b - c;
    e = b * c + g;
    return d + e;
}

int main() {
    test(1,2,3,4);
    return 0;
}
```

partial redundancy elimination (PRE)
Eliminates unnecessary and redundant expressions on some paths through of the code

`-fcode-hoisting`

-O2 optimization level or higher, and on -Os.
The objdumpt before/after the optimization will be:

gcc hosting.c -O1

```asm
0000000000001129 <test>:
  1129:  89 d0  mov %edx,%eax
  112b:  89 f2  mov %esi,%edx
  112d:  29 c2  sub %eax,%edx
  112f:  85 ff  test %edi,%edi
  1131:  74 05  je  1138 <test+0xf>
  1133:  89 f2  mov %esi,%edx
  1135:  0f af d0  imul %eax,%edx
  1138:  0f af f0  imul %eax,%esi
  113b:  8d 04 0e  lea (%rsi,%rcx,1),%eax
  113e:  01 d0  add %edx,%eax
  1140:  c3  retq
```

gcc hosting.c -O2

```asm
0000000000001140 <test>:
  1140:  89 f0  mov %esi,%eax
  1142:  29 d6  sub %edx,%esi
  1144:  0f af c2  imul %edx,%eax
  1147:  85 ff  test %edi,%edi
  1149:  0f 45 f0  cmovne %eax,%esi
  114c:  01 c1  add %eax,%ecx
  114e:  8d 04 31  lea (%rcx,%rsi,1),%eax
  1151:  c3  retq
  1152:  66 2e 0f 1f 84 00 00  nopw %cs:0x0(%rax,%rax,1)
  1159:  00 00 00
  115c:  0f 1f 40 00  nopl 0x0(%rax)
```
Optimizations to the generic mathematical functions

GNU C Library recently present a set of optimizations to the generic mathematical functions. One of these functions is sincosf(). Which calculate sine and cosine of the same angle x

```c
#define _GNU_SOURCE
#include <math.h>
#include <stdio.h>

int main(int argc, const char * argv[]){
    float value = 0.5;
    
    float _cosine;
    float _sine;
    
sincosf(value, &_sine, &_cosine);
    
    printf("The sine of %f is %f\n", value,_sine);
    printf("The cosine of %f is %f\n", value,_cosine);
    
    return 0;
}
```
• The core algorithm of this function is in s_sincosf.c file and syscall_polly.h where it uses the sincosf_poly function.
• This function computes the sine and cosine using the polynomial P algorithm

\[
f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f^{(3)}(a)}{3!}(x-a)^3 + \cdots.
\]
• There are some advantages to using the polynomial method.
• First, the memory requirements necessary to implement such polynomials are quite small.
• Also, polynomials only require multiplication, addition, and subtraction of floating-point numbers which normally take very few CPU cycles for processors with floating point cores.
• The implementation of this algorithm for the function is part of the sincosf_poly.h
Developer experience
Smarter fix-it hints

```c
#include<stdio.h>
int main(){
    int color;
    printf("Hello World\n");
    printf("%d",colour);
    return 0;
}
```

```
$ gcc hello.c -o hello -fdiagnostics-parseable-fixits
hello.c: In function ‘main’:
hello.c:6:17: error: ‘colour’ undeclared (first use in this function); did you mean
    printf("%d",colour);
      ^~~~~~~
    color
fix-it:"hello.c":{6:17-6:23}:"color"
hello.c:6:17: note: each undeclared identifier is reported only once for each functi
```
-fdiagnostics-generate-patch

```c
hello.c: In function ‘main’:
hello.c:6:17: error: ‘colour’ undeclared (first use in this function); did you mean
    printf("%d",colour);
    ^~~~~~
    color

hello.c:6:17: note: each undeclared identifier is reported only once for each function
--- hello.c
+++ hello.c
@@ -3,6 +3,6 @@
int main(){
    int color;
    printf("Hello World\n");
-    printf("%d",colour);
+    printf("%d",color);
    return 0;
}
```
-fdiagnostics-format=FORMAT where it is possible to select a different format for printing diagnostics.

FORMAT is `text' or 'json'.

The next example takes the previous code with error to print the report in json format:

```
gcc diagnostic.c -fdiagnostics-format=json

[
{"kind": "warning", "option": "-Wint-conversion", "children": [{"kind": "note", "locations": [{"caret": {"line": 3, "file": "diagnostic.c", "column": 14}, "start": {"line": 3, "file": "diagnostic.c", "column": 10}}], "message": "expected 'int' but argument is of type 'char *'"}], "locations": [{"caret": {"line": 8, "file": "diagnostic.c", "column": 6}, "label": "char *"}], "message": "passing argument 1 of 'foo' makes integer from pointer without a cast"}
```
```c
int foo (int how) {
    switch (how) {
        case 2: how = 205; break;
        case 3: how = 305; break;
        case 4: how = 405; break;
        case 5: how = 505; break;
        case 6: how = 605; break;
    }
    return how;
}

void main(){
    int var = 3;
    foo(var);
}
```

100 * how + 5 (for this example)
0000000000001129 <foo>:
    1129: 83 ff 06    cmp $0x6,%edi
    112c: 77 34    ja 1162 <foo+0x39>
    112e: 89 fa    mov %edi,%edx
    1130: 48 8d 0d cd 0e 00 00    lea 0xecd(%rip),%rcx  # 2004
  <_IO_stdin_used+0x4>
    1137: 48 63 04 91 movslq (%rcx,%rdx,4),%rax
    113b: 48 01 c8    add %rcx,%rax
    113e: ff e0    jmpq *%rax
    1140: b8 cd 00 00 00 mov $0xc0d,%eax
    1145: c3    retq
    1146: b8 31 01 00 00 mov $0x131,%eax
    114b: eb f8    jmp 1145 <foo+0x1c>
    114d: b8 95 01 00 00 mov $0x195,%eax
    1152: eb f1    jmp 1145 <foo+0x1c>
    1154: b8 f9 01 00 00 mov $0x1f9,%eax
    1159: eb ea    jmp 1145 <foo+0x1c>
    115b: b8 5d 02 00 00 mov $0x25d,%eax
    1160: eb e3    jmp 1145 <foo+0x1c>
    1162: 89 f8    mov %edi,%eax
    1164: eb df    jmp 1145 <foo+0x1c>

0000000000001129 <foo>:
    1129: 89 f8    mov %edi,%eax
    112b: 8d 57 fe    lea -0x2(%rdi),%edx
    112e: 83 fa 04    cmp $0x4,%edx
    1131: 77 06    ja 1139 <foo+0x10>
    1133: 6b c7 64    imul $0x64,%edi,%eax
    1136: 83 c0 05    add $0x5,%eax
    1139: c3    retq
And more...

- `fno-diagnostics-show-line-numbers`
- `Walloca-larger-than`
- `getcpu()` system call

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <sched.h>

int main(){
    unsigned int *cpu;
    unsigned int *node;
    int ret;

    ret = getcpu(cpu,node);

    printf("cpu : %d\n",(int *)cpu);
    printf("node : %d\n",(int *)node);

    return ret;
}
```
“The open source community behind the development of GNU toolchain projects, use the innovation to propel the core of the technology we use every day.”
Clear Linux* OS as example:

- GCC-7-importance-cutting-edge-compiler
  - [https://clearlinux.org/news-blogs/gcc-7-importance-cutting-edge-compiler](https://clearlinux.org/news-blogs/gcc-7-importance-cutting-edge-compiler)

- GCC 8- Transitioning to a new compiler

- GCC 9 – GLIBC-30 Toolchains as fresh wind in the sails of a new tech world
Back up
The address of secretFunction is 080484b2 in hex:

```
080484b2 <secretFunction>:
```

Then 28hex (40 decimal) bytes are reserved for the local variables of echo function.

```
080484db <echo>:
  80484db: 55                   push %ebp
  80484dc: 89 e5                 mov %esp, %ebp
  80484de: 83 ec 28               sub $0x28, %esp
  80484e1: 83 ec 0c               sub $0xc, %esp
```

The address of buffer starts 1c (28 decimal) bytes before %ebp.

```
08484f1: 83 ec 08                sub $0x8, %esp
08484f4: 8d 45 e4               lea -0x1c(%ebp), %eax
08484f7: 50                      push %eax
```
Another way to protect code against these kind of attacks is `mzero-caller-saved-reg`=[skip | used | all].

This option clears caller-saved general registers upon function return.

This is intended to make threats such as ROP, COP, and JOP attacks much harder.
Detect buffer overflow and invalid memory accesses

```c
void f (int n) { 
    char *d;
    if (n < 1025) 
        d = alloca (n);
    else 
        d = malloc (n);
...
}
```

When compiled with the new flag

-`-Walloca-larger-than=1024`

GCC shows this warning:

```
warning: argument to 'alloca' may be too large due to conversion from 'int' to 'long unsigned int' [-Walloca-larger-than=]
```