Adventure with Stack Smashing Protector (SSP)

Introduction.

I was heavily playing with Stack Smashing Protector a few years ago. Some of my research (observation) I decided to publish on phrack magazine but not everything. Two years ago my professional life moved to the Windows environment and unfortunately I didn’t have time to play with UNIX world as much as before. One weekend I decided to reanalyze SSP code again and this write-up is describing a few of my observations I’ve made during the work…

... which can be shortly summarized as (details can be found at “Random ideas” section):

Not security related…

1. We can change program’s name (from SSP perspective) via overwriting memory region where pointer to "argv[0]" points to.
2. We can crash Stack Smashing Protector code in many ways:
   a. Via corrupting memory region pointed by "__environ" variable.
   b. Via setting "LIBC_FATAL_STDERR_" to the edge of valid addresses.
   c. Via forcing "alloca()" to fail – e.g. stack exhaustion.
   d. There is one more bug which I’m analyzing more comprehensively at point 4. It may indirectly force SSP to crash. It exists in DWARF stack (state) machine which is responsible for gathering information about the stack trace("__backtrace()") and prints it.
3. We can slightly control SSP’s execution flow. (Un)Fortunately it doesn’t have any influence for the main execution (what about security?). Following scenarios are possible:
   a. Force SSP to open "/dev/tty"
   b. Force SSP **not** to open "/dev/tty" and assign to the "fd" descriptor "STDERR_FILENO" value:

```
#define STDERR_FILENO   2   /* Standard error output. */
```
   c. Crash SSP via 2b. scenario
4. We can crash indirectly SSP via unwinding algorithm (read-AV or we can be killed by "gcc_unreachable" or "gcc_assert" function) – DWARF stack (state) machine:
   a. Simulate FDE object was not found
   b. Simulate FDE object was found.

Somehow security related… (look at “Random ideas” section for details):

1. We can force SSP to allocate a lot of memory and cause Denial of Service via Resource Exhaustion attack.
2. Theoretical Information leak:
   a. Stack cookie information leak.
   b. Any kind of information leak
   c. File corruption.
Stack Smashing Protector (SSP) a.k.a ProPolice under the microscope.

GNU Compiler Collection (GCC) includes SSP implementation and is the most commonly used one. Let's analyze the full code chain called during SSP execution. If corruption on the stack is detected (stack canary is not valid), the following function is called:

```
"debug/stack_chk_fail.c"
void
__attribute__ ((noreturn))
__stack_chk_fail (void)
{
  __fortify_fail ("stack smashing detected");
}
```

Nothing fancy here so let's move forward:

```
"debug/fortify_fail.c"
void
__attribute__ ((noreturn))
__fortify_fail (msg)
    const char *msg;
{
  /* The loop is added only to keep gcc happy. */
  while (1)
    __libc_message (2, "*** %s ***: %s terminated\n", 
                   msg, __libc_argv[0] ?: "<unknown>");
}
libc_hidden_def (__fortify_fail)
```

First discovery in the very early stage is `__libc_argv[0]` cannot be trusted, because the memory area where it's pointing to can be modified at the time of crash (each pointer on the stack may be corrupted). Moving forward:

```
"sysdeps/unix/sysv/linux/libc_fatal.c"
/* Abort with an error message. */
void
__libc_message (int do_abort, const char *fmt, ...)
{
  va_list ap;
  va_list ap_copy;
  int fd = -1;

  va_start (ap, fmt);
  va_copy (ap_copy, ap);

  #ifdef FATAL_PREPARE
  FATAL_PREPARE;
  #endif
```
/* Open a descriptor for /dev/tty unless the user explicitly requests errors on standard error. */
const char *on_2 = __libc_secure_getenv("LIBC_FATAL_STDERR_");
if (on_2 == NULL || *on_2 == '\0')
    fd = open_not_cancel_2(_PATH_TTY, O_RDWR | O_NOCTTY | O_NDELAY);

if (fd == -1)
    fd = STDERR_FILENO;

struct str_list *list = NULL;
int nlist = 0;

const char *cp = fmt;
while (*cp != '\0')
{
    /* Find the next "%s" or the end of the string. */
    const char *next = cp;
    while (next[0] != '%' || next[1] != 's')
    {
        next = __strchrnul(next + 1, '%');
        if (next[0] == '\0')
            break;
    }

    /* Determine what to print. */
    const char *str;
    size_t len;
    if (cp[0] == '%' && cp[1] == 's')
    {
        str = va_arg(ap, const char *);
        len = strlen(str);
        cp += 2;
    }
    else
    {
        str = cp;
        len = next - cp;
        cp = next;
    }

    struct str_list *newp = alloca(sizeof(struct str_list));
    newp->str = str;
    newp->len = len;
    newp->next = list;
    list = newp;
    ++nlist;
}

bool written = false;
if (nlist > 0)
{  
    struct iovec *iov = alloca (nlist * sizeof (struct iovec));
    ssize_t total = 0;

    for (int cnt = nlist - 1; cnt >= 0; --cnt) {
        iov[cnt].iov_base = (void *) list->str;
        iov[cnt].iov_len = list->len;
        total += list->len;
        list = list->next;
    }

    INTERNAL_SYSCALL_DECL (err);
    ssize_t cnt;

    do
        cnt = INTERNAL_SYSCALL (writev, err, 3, fd, iov, nlist);
    while (INTERNAL_SYSCALL_ERROR_P (cnt, err)
        && INTERNAL_SYSCALL_ERRNO (cnt, err) == EINTR);

    if (cnt == total)
        written = true;

    if (do_abort)
        {  
            total = ((total + 1 + GLRO(dl_pagesize) - 1)
                & ~(GLRO(dl_pagesize) - 1));
            struct abort_msg_s *buf = __mmap (NULL, total,
                PROT_READ | PROT_WRITE,
                MAP_ANON | MAP_PRIVATE, -1, 0);
            if (__builtin_expect (buf != MAP_FAILED, 1))
                {  
                    buf->size = total;
                    char *wp = buf->msg;
                    for (int cnt = 0; cnt < nlist; ++cnt)
                        wp = mempcpy (wp, iov[cnt].iov_base, iov[cnt].iov_len);
                    *wp = '\0';

                    /* We have to free the old buffer since the application
                       might catch the SIGABRT signal. */
                    struct abort_msg_s *old = atomic_exchange_acq
                        (&__abort_msg,
                            buf);
                    if (old != NULL)
                        __munmap (old, old->size);
                }
        }

    va_end (ap);

    /* If we had no success writing the message, use syslog. */
What is interesting in this function? At first, before function `abort()` is called a lot of code (too much ;) is executed. This is not amazing idea, because corrupted process cannot be trusted, and execution of any code (especially the code which relies on any pointer[s]) is unexpected. Let's look closer for the following line:

```
const char *on_2 = __libc_secure_getenv ("LIBC_FATAL_STDERR_");
```

Which essentially executes:

```
"stdlib/secure-getenv.c"
char *
__libc_secure_getenv (name)
{
const char *name;
```
return __libc_enable_secure ? NULL : getenv (name);
}

Moving forward:

"stdlib/getenv.c"
char *
getenv (name)
    const char *name;
{
    size_t len = strlen (name);
    char **ep;
    uint16_t name_start;

    if (__environ == NULL || name[0] == '\0')
        return NULL;

    if (name[1] == '\0')
    {
        /* The name of the variable consists of only one character.
         * Therefore
         * the first two characters of the environment entry are this
         * character
         * and a '=' character. */
        #if __BYTE_ORDER == __LITTLE_ENDIAN || !_STRING_ARCH_unaligned
            name_start = ('=' << 8) | *(const unsigned char *) name;
        #else
            # if __BYTE_ORDER == __BIG_ENDIAN
                name_start = '=' | ((*(const unsigned char *) name) << 8);
            # else
                #error "Funny byte order."
            # endif
        #endif
        for (ep = __environ; *ep != NULL; ++ep)
        {
            #if _STRING_ARCH_unaligned
                uint16_t ep_start = *(uint16_t *) ep;
            #else
                uint16_t ep_start = (((unsigned char *) *ep)[0] | (((unsigned char *) *ep)[1] << 8));
            #endif
            if (name_start == ep_start)
                return &(*ep)[2];
        }
        #endif

    else
    {
        #if _STRING_ARCH_unaligned
            name_start = *(const uint16_t *) name;
        #else
            name_start = (((const unsigned char *) name)[0] | (((const unsigned char *) name)[1] << 8));
        #endif
    }
}
Essentially this function goes through the environment block/array (pointer to the pointers) and checks if first two characters are equivalent to the arguments'. If yes it checks the rest of the string. The last step of this function is to verify if character '=' exists and in this case address to the next byte is returned.
Again, this code relies on the pointers which can be corrupted. We are able to easily crash this code here when it references blocks from the environment block/array (which I will prove later).
Additionally, bytes after '=' character is not verified and just simple address is returned.

After executing `getenv()` function, following code is executed:

```c
/* Open a descriptor for /dev/tty unless the user explicitly requests errors on standard error. */
const char *on_2 = __libc_secure_getenv("LIBC_FATAL_STDERR_");
if (on_2 == NULL || *on_2 == '\0')
    fd = open_not_cancel_2(_PATH_TTY, O_RDWR | O_NOCTTY | O_NDELAY);
    if (fd == -1)
        fd = STDERR_FILENO;
```

If not NULL is return by `__libc_secure_getenv()`, the code references the address and verifies if it is pointing to the NULL. When both checks are passed function `open_not_cancel_2()` is called. In the end just default `open()` function is executed. What is _PATH_TTY?

```c
#define _PATH_TTY   "/dev/tty"
```

We are able to execute (control) 3 possible scenarios:
1. Force SSP to open /dev/tty.
2. Force SSP to NOT open /dev/tty and just assign to the 'fd' descriptor value STDERR_FILENO which means:

```c
#define  STDERR_FILENO   2   /* Standard error output.  */
```

3. Crash SSP via setting LIBC_FATAL_STDERR_ to the edge of valid addresses

SSP can be crashed in much easier (than 3rd scenario) way just by completely messing up environmental block (variable “__environ”).

Another interesting piece of code dynamically allocates memory via alloca() function. For example:

```c
...  
struct str_list *newp = alloca (sizeof (struct str_list));
newp->str = str;
newp->len = len;
newp->next = list;
list = newp;
++nlist;
...
struct iovec *iov = alloca (nlist * sizeof (struct iovec));
ssize_t total = 0;
for (int cnt = nlist - 1; cnt >= 0; --cnt) 
{ 
  iov[cnt].iov_base = (void *) list->str;
  iov[cnt].iov_len = list->len;
  total += list->len;
  list = list->next;
}
...
```

Again, this code is dangerous because SIGSEGV can be received in some specific scenarios. Let me quote following information from the GNU:

"Normally, gcc(1) translates calls to alloca() with inlined code. This is not done when either the -ansi, -std=c89, -std=c99, or the -std=c11 option is given and the header <alloca.h> is not included. Otherwise (without an -ansi or -std=c* option) the glibc version of <stdlib.h> includes <alloca.h> and that contains the lines:

```c
#ifdef  __GNUC__
#define alloca(size)   __builtin_alloca (size)
#endif
```

with messy consequences if one has a private version of this function."
The fact that the code is inlined means that it is impossible to take the address of this function, or to change its behavior by linking with a different library.

The inlined code often consists of a single instruction adjusting the stack pointer, and does not check for stack overflow. Thus, there is no NULL error return."

and continuing:

"There is no error indication if the stack frame cannot be extended. (However, after a failed allocation, the program is likely to receive a SIGSEGV signal if it attempts to access the unallocated space.)"

OK, so now let's analyze something more interesting. Let's look for the following code:

--- CUT ---

```c
struct str_list *list = NULL;
int nlist = 0;

const char *cp = fmt;
while (*cp != '\0')
{
...
...
/* Determine what to print. */
const char *str;
size_t len;
if (cp[0] == '%' && cp[1] == 's')
{
  str = va_arg (ap, const char *);
[1]  len = strlen (str);
  cp += 2;
}
...
...
bool written = false;
if (nlist > 0)
{
  struct iovec *iov = alloca (nlist * sizeof (struct iovec));
  ssize_t total = 0;
  for (int cnt = nlist - 1; cnt >= 0; --cnt)
  {
    iov[cnt].iov_base = (void *) list->str;
    iov[cnt].iov_len = list->len;
[2]    total += list->len;
    list = list->next;
```
I've added a few tags with numbers. Let's start from the [1] tag. If function format includes any string argument ("%s") sequence it will be extracted (va_arg()), assigned to the "str" variable, length will be calculated (via strlen()) and assigned to the "len" variable. This scenario will be executed for every "%s" formatter.

This is very important, because we can control one of the argument to the string formatter. If you look closer at the beginning of this write-up you will realize why. Short reminder:

```
__libc_message (2, "*** %s ***: %s terminated\n",
    msg, __libc_argv[0] ?: "<unknown>");
```

We can overflow memory where pointer "__libc_argv[0]" points to and change the displayed name of crashed application. What is even more important, we can change behavior of the "len" and "str" variables. In fact in some way we can control "len" variable.
Next, if we move to the [2] tag, you may discover that we can indirectly control "total" variable as well. This variable is updated each passing of the loop. The code inside just go through the list (built before) and for every substring calculates their length and updates "iov" "database". The "total" variable keeps the full length calculated from each substring.

At line [3], variable "total" is recalculated (aligned to the page) and at line [4] used as a size argument to the "__mmap" function (dynamic memory allocation). Most of you probably realize that we can control how much memory will be dynamically allocated. Next (line [5]) "total" is kept in the newly allocated buffer’s metadata. At line [6] dynamic buffer is assigned to the temporary pointer. Line [7] is inside of the loop which “extracts” previously created "iov" "database" and copy all data to the newly allocated memory.

What can we get via this scenario? We can force SSP to allocate big chunk of memory which later will be referenced and some data copied. This may results with small resources exhaustion attack.

The last stage of "__libc_message" function is to execute following code in case "do_abort" is declared (which is in our situation):

```c
if (do_abort)
{
  if (do_abort > 1 && written)
  {
    void *addrs[64];
    #define naddrs (sizeof (addrs) / sizeof (addrs[0]))
    int n = __backtrace (addrs, naddrs);
    if (n > 2)
    {
      #define strnsize(str) str, strlen (str)
      #define writestr(str) write_not_cancel (fd, str)
      writestr (strnsize ("======= Backtrace: =========
      " ));
      __backtrace_symbols_fd (addrs + 1, n - 1, fd);
      writestr (strnsize ("======= Memory map: =======
      " ));
      int fd2 = open_not_cancel_2 ("/proc/self/maps", O_RDONLY);
      char buf[1024];
      ssize_t n2;
      while ((n2 = read_not_cancel (fd2, buf, sizeof (buf))) > 0)
      {
        if (write_not_cancel (fd, buf, n2) != n2)
          break;
        close_not_cancel_no_status (fd2);
      }
    }
  }
  /* Terminate the process. */
  abort ();
}
```
The most important and interesting is "__backtrace" function. It’s very easy to crash the SSP code on read access violation (AV) in the depth of function calls through the "__backtrace". Backtracing in gcc heavily use DWARF. Before we analyze source code in details, it’s good time to describe a bit more what and how is DWARF used for...

"Debugging With Attributed Record Formats" – DWARF.

DWARF is a debugging format used to describe programs in C and other similar programming languages. It is most widely associated with the ELF object format but it has been used with other object file formats. Additionally gcc uses DWARF mechanism for stack unwinding in general, and also for C++ exception handling.

To handle an exception, the stack must be unwound. Unfortunately this problem can’t be shortened just to the walk the call stack following return address pointers to find all call frames. Mainly because this information are not enough to restore execution to an exception handler as well as this process does not respect register state. To solve this problems Call-Frame Information section (unwinding information) of the DWARF standard has been adopted (with some changes) for exception handling.

Quoting the excellent research paper "Exploiting the hard-working dwarf" by James Oakley and Sergey Bratus:

"Conceptually, what this unwinding information describes is a large table. The rows of the table correspond to machine instructions in the program text, and the columns correspond to registers and Canonical Frame Address (CFA). Each row describes how to restore the machine state (the values of the registers and CFA) for every instruction at the previous call frame as if control were to return up the stack from that instruction. DWARF allows for an arbitrary number of registers, identified merely by number. It is up to individual ABIs to define a mapping between DWARF register numbers and the hardware registers. The DWARF registers are not required to map to actual hardware registers, but may be used internally, as is often done with a DWARF register for the return address. Each cell of this table holds a rule detailing how the contents of the register will be restored for the previous call frame. DWARF allows for several types of rules, and the curious reader is invited to find them in the DWARF standard. Most registers are restored either from another register or from a memory location accessed at some offset from the CFA.

We note that this table, if constructed in its entirety, would be absurdly large, larger than the text of the program itself. There are many empty cells and many duplicated entries in columns. Much of the DWARF call frame information standard is essentially a compression technique, allowing to provide sufficient information at runtime to build parts of the table as needed without the full, prohibitively large, table ever being built or stored. This compression is performed by introducing the concept of Frame Description Entities (FDEs) and DWARF instructions. An FDE corresponds to a logical block of program text and describes how unwinding may be done from within that block. Each FDE contains a series of DWARF instructions. Each instruction either specifies
one of the column rules (registers) as from our table above or
specifies which text locations the register rules apply to."

More details may be found in DWARF Debugging Standard Website.

It is also worthiest to understand how exception handler is encoded and handled because conception of
backtracing in gcc is very similar (to be honest, almost the same excluding call to the EH – personality
routine) to the backtracing which we want to analyze ("__backtrace" function). I want to quote again
James Oakley’s and Sergey Bratus’s paper:

"DWARF is designed as a debugging format, where the debugger is in
control of how far to unwind the stack. DWARF therefore does not
provide any mechanism to govern halting the unwinding process. What
does provide is the means for augmentation to the standard.
Certain DWARF data structures include an augmentation string, the
contents of which are implementation defined, allowing a DWARF
producer to communicate to a compatible DWARF consumer information
not controlled by the standard. The augmentations to be used on
Linux and x86 64 are well-defined. These augmentations allow a
language-specific data area (LSDA) and personality routine to be
associated with every FDE.

When unwinding an FDE, the exception handling process is required
to call the personality routine associated with the FDE. The
personality routine interprets the LSDA and determines if a handler
for the exception has been found. The actual contents of the LSDA
are not defined by any standard, and two separate compilation units
originally written in different languages and using different LSDA
formats may coexist in the same program, as they will be served by
separate personality routines.

The result of these design decisions is that the encoding of where
exception handlers are located and what type of exceptions they
handle is mostly nonstandardized. The best known source of
information on the format used by gcc is the verbose assembly code
generated by gcc. (...) In an ELF binary, the section .gcc except
table contains the LSDAs. In the environment we are concerned with,
an LSDA breaks the text region described by the corresponding FDE
into call sites. Each call site corresponds to code within a try
block (to use C++ terminology) and has a pointer to a chain of C++
typeinfo descriptors. These objects are used by the personality
routine to determine whether the thrown exception can be handled
in the current frame.

(...) During Exception Process, libgcc computes the machine state as a
result of the unwinding, directly restores the necessary registers,
and then returns into the handler code, which is known as the
landing pad. We note that, at least in current (4.5.2) gcc
implementations, this means that at the time execution is first
returned to the handler code, the data from which the registers
were restored will still be present below the stack pointer until
it is overwritten"
Now we have solid knowledge about DWARF itself and some expectation how gcc should use it in backtracing algorithm. Let’s analyze following code:

```
"sysdeps/x86_64/backtrace.c"
int
__backtrace (array, size)
  void **array;
  int size;
{
  struct trace_arg arg = { .array = array, .cfa = 0, .size = size, .cnt = -1 };
  #ifdef SHARED
  __libc_once_define (static, once);
  __libc_once (once, init);
  if (unwind_backtrace == NULL)
    return 0;
  #endif
  if (size >= 1)
    unwind_backtrace (backtrace_helper, &arg);
  /* _Unwind_Backtrace seems to put NULL address above _start. Fix it up here. */
  if (arg.cnt > 1 && arg.array[arg.cnt - 1] == NULL)
    --arg.cnt;
  return arg.cnt != -1 ? arg.cnt : 0;
}
weak_alias (__backtrace, backtrace)
libc_hidden_def (__backtrace)
```

Where:

```
static _Unwind_Reason_Code (*unwind_backtrace) (_Unwind_Trace_Fn, void *);
...
static void *libgcc_handle;
...
  libgcc_handle = __libc_dlopen ("libgcc_s.so.1");
...
  unwind_backtrace = __libc_dlsym (libgcc_handle, "_Unwind_Backtrace");
...
```

Before we move to the "_Unwind_Backtrace" function, let’s see helper function passed as an argument to it:

```
static _Unwind_Reason_Code
backtrace_helper (struct _Unwind_Context *ctx, void *a)
```
{  
    struct trace_arg *arg = a;

    /* We are first called with address in the __backtrace function.  
    Skip it. */
    if (arg->cnt != -1)  
    {  
        arg->array[arg->cnt] = (void *) unwind_getip (ctx);

        /* Check whether we make any progress. */
        _Unwind_Word cfa = unwind_getcfa (ctx);

        if (arg->cnt > 0 && arg->array[arg->cnt - 1] == arg->array[arg->cnt]  
            && cfa == arg->cfa)  
            return _URC_END_OF_STACK;

        arg->cfa = cfa;
    }

    if (++arg->cnt == arg->size)  
        return _URC_END_OF_STACK;

    return _URC_NO_REASON;
}

Where:

unwind_getip = __libc_dlsym (libgcc_handle, "_Unwind_GetIP");
unwind_getcfa = (__libc_dlsym (libgcc_handle, "_Unwind_GetCFA")  
?: dummy_getcfa);

inline _Unwind_Ptr
_Unwind_GetIP (struct _Unwind_Context *context)  
{  
    return (_Unwind_Ptr) context->ra;
}

_Unwind_Word
_Unwind_GetCFA (struct _Unwind_Context *context)  
{  
    return (_Unwind_Ptr) context->cfa;
}

In short, helper function is responsible for checking if there is any “progress” in stack unwinding by  
analyzing CFA. It also prevents from the looping around the same frames.
Returning to the main unwinding function:

"libgcc/unwind.inc"
/* Perform stack backtrace through unwind data. */

_Unwind_Reason_Code LIBGCC2_UNWIND_ATTRIBUTE
_Unwind_Backtrace(_Unwind_Trace_Fn trace, void * trace_argument)  
{
In short this function is responsible for setting up current frame state ("fs" variable) based on current context. After that "context" is updated for the next frame and parsing starts again. This infinitive loop will break if algorithm detects that current frame is the last one ("_URC_END_OF_STACK"). Let’s move to the most important function in this algorithm:

"libgcc/unwind-dw2.c"
/* Given the _Unwind_Context CONTEXT for a stack frame, look up the FDE for its caller and decode it into FS. This function also sets the args_size and ldsa members of CONTEXT, as they are really information about the caller's frame. */

static _Unwind_Reason_Code
uw_frame_state_for (struct _Unwind_Context *context, _Unwind_FrameState *fs)
{
    const struct dwarf_fde *fde;
    const struct dwarf_cie *cie;
    const unsigned char *aug, *insn, *end;

    /* Set up fs to describe the FDE for the caller of context. */
    code = uw_frame_state_for (&context, &fs);
    if (code != _URC_NO_REASON && code != _URC_END_OF_STACK)
        return _URC_FATAL_PHASE1_ERROR;

    /* Call trace function. */
    if ((*trace) (&context, trace_argument) != _URC_NO_REASON)
        return _URC_FATAL_PHASE1_ERROR;

    /* We're done at end of stack. */
    if (code == _URC_END_OF_STACK)
        break;

    /* Update context to describe the same frame as fs. */
    uw_update_context (&context, &fs);
}

return code;
}
memset (fs, 0, sizeof (*fs));
context->args_size = 0;
context->lsda = 0;

if (context->ra == 0)
    return _URC_END_OF_STACK;

fde = _Unwind_Find_FDE (context->ra + _Unwind_IsSignalFrame (context) - 1, &context->bases);
if (fde == NULL)
{
    #ifdef MD_FALLBACK_FRAME_STATE_FOR
        /* Couldn't find frame unwind info for this function. Try a
target-specific fallback mechanism. This will
necessarily
        not provide a personality routine or LSDA. */
        return MD_FALLBACK_FRAME_STATE_FOR (context, fs);
    #else
        return _URC_END_OF_STACK;
    #endif
}

fs->pc = context->bases.func;

cie = get_cie (fde);
insn = extract_cie_info (cie, context, fs);
if (insn == NULL)
    /* CIE contained unknown augmentation. */
    return _URC_FATAL_PHASE1_ERROR;

/* First decode all the insns in the CIE. */
end = (const unsigned char *) next_fde ((const struct dwarf_fde *) cie);
execute_cfa_program (insn, end, context, fs);

/* Locate augmentation for the fde. */
aug = (const unsigned char *) fde + sizeof (*fde);
aug += 2 * size_of_encoded_value (fs->fde_encoding);
insn = NULL;
if (fs->saw_z)
{
    _uleb128_t i;
    aug = read_uleb128 (aug, &i);
    insn = aug + i;
}
if (fs->lsda_encoding != DW_EH_PE_omit)
{
    _Unwind_Ptr lsda;

    aug = read_encoded_value (context, fs->lsda_encoding, aug, &lsda);
If return address of current context is 0 (which is indicator for the end of stack) function immediately returns. Otherwise complicated "Unwind_Find_FDE" function is called. The main goal of it is to find FDE object based on current context and return address:

const fde *
_Unwind_Find_FDE (void *pc, struct dwarf_eh_bases *bases)
{
    struct object *ob;
    const fde *f = NULL;

    init_object_mutex_once ();
    __gthread_mutex_lock (&object_mutex);

    /* Linear search through the classified objects, to find the one containing the pc. Note that pc_begin is sorted descending, and we expect objects to be non-overlapping. */
    for (ob = seen_objects; ob; ob = ob->next)
        if (pc >= ob->pc_begin)
        {
            f = search_object (ob, pc);
            if (f)
                goto fini;
            break;
        }

    /* Classify and search the objects we've not yet processed. */
    while ((ob = unseen_objects))
    {
        struct object **p;

        unseen_objects = ob->next;
        f = search_object (ob, pc);

        /* Insert the object into the classified list. */
        for (p = &seen_objects; *p ; p = &(*p)->next)
            if ((*p)->pc_begin < ob->pc_begin)
                break;
        ob->next = *p;

    return (void *) lsda;
}

/* Then the insns in the FDE up to our target PC. */
if (insn == NULL)
    insn = aug;
end = (const unsigned char *) next_fde (fde);
execute_cfa_program (insn, end, context, fs);
return _URC_NO_REASON;
}
This function is responsible to find out FDE object based on current return address read from the frame (which can be fully controllable by us). The key function is "search_object":

```c
static const fde *
search_object (struct object* ob, void *pc) {
  /* If the data hasn't been sorted, try to do this now. We may have
     more memory available than last time we tried. */
  if (! ob->s.b.sorted)
      init_object (ob);

  /* Despite the above comment, the normal reason to get here is
     that we've not processed this object before. A quick range
     check is in order. */
  if (pc < ob->pc_begin)
      return NULL;
}
```
In general, different type of searching algorithm is executed ("binary_search_mixed_encoding_fdes", "binary_search_unencoded_fdes", "binary_search_single_encoding_fdes", "linear_search_fdes"). Each of the function depends on return address as a range of search. Because at stack overflow bugs we fully control return address we can point it to the memory where special prepared bytes can be recognized as correct (or not) and specific existing in the process FDE object can be chosen. Next based on what "_Unwind_Find_FDE" found (or not) CIE object may be calculated. Quoting gcc internal source code comments:

/*
 CIE - Common Information Element
 FDE - Frame Descriptor Element

 There is one per function, and it describes where the function code is located, and what the register lifetimes and stack layout are within the function.
 The data structures are defined in the DWARF specification, although not in a very readable way (see LITERATURE).

 Every time an exception is thrown, the code needs to locate the FDE for the current function, and starts to look for exception regions from that FDE. This works in a two-level search:
 a) in a linear search, find the shared image (i.e. DLL)
containing the PC
b) using the FDE table for that shared object, locate the FDE
using binary search (which requires the sorting). */

This is quite interesting satiation because we can choose which code path to execute. Let’s at first simulate (analyze) easier one – none of the FDE objects was found. In this case following lines are executed:

```c
if (fde == NULL)
{
    #ifdef MD_FALLBACK_FRAME_STATE_FOR
        /* Couldn't find frame unwind info for this function. Try a
target-specific fallback mechanism. This will necessarily
not provide a personality routine or LSDA. */
    return MD_FALLBACK_FRAME_STATE_FOR (context, fs);
    #else
    return _URC_END_OF_STACK;
    #endif
}
```

"MD_FALLBACK_FRAME_STATE_FOR" is defined by default so:

```c
obj-x86_64-redhat-linux/x86_64-redhat-linux/libgcc/md-unwind-support.h:
#define MD_FALLBACK_FRAME_STATE_FOR x86_64_fallback_frame_state
```

I’m using 64 bits VM for this research so this case will be analyzed. Fortunately there is not much differences between them:

```c
static _Unwind_Reason_Code
x86_64_fallback_frame_state (struct _Unwind_Context *context,
            _Unwind_FrameState *fs)
{
    unsigned char *pc = context->ra;
    struct sigcontext *sc;
    long new_cfa;

    /* movq $__NR_rt_sigreturn, %rax ; syscall. */
    #ifdef __LP64__
    #define RT_SIGRETURN_SYSCALL 0x050f0000000fc0c7ULL
    #else
    #define RT_SIGRETURN_SYSCALL 0x050f40000201c0c7ULL
    #endif
    if (*(unsigned char *)(pc+0) == 0x48
        && *(unsigned long long *)(pc+1) == RT_SIGRETURN_SYSCALL)
    {
        struct ucontext *uc_ = context->cfa;
```
/* The void * cast is necessary to avoid an aliasing warning. The aliasing warning is correct, but should not be a problem because it does not alias anything. */
	sc = (struct sigcontext *) (void *) &uc_->uc_mcontext;
}
else
	return _URC_END_OF_STACK;

new_cfa = sc->rsp;
fs->regs.cfa_how = CFA_REG_OFFSET;
/* Register 7 is rsp */
fs->regs.cfa_reg = 7;
fs->regs.cfa_offset = new_cfa - (long) context->cfa;

/* The SVR4 register numbering macros aren't usable in libgcc. */
	fs->regs.reg[0].how = REG_SAVED_OFFSET;
	fs->regs.reg[0].loc.offset = (long)&sc->rax - new_cfa;
	fs->regs.reg[1].how = REG_SAVED_OFFSET;
	fs->regs.reg[1].loc.offset = (long)&sc->rdx - new_cfa;
	fs->regs.reg[2].how = REG_SAVED_OFFSET;
	fs->regs.reg[2].loc.offset = (long)&sc->rcx - new_cfa;
	fs->regs.reg[3].how = REG_SAVED_OFFSET;
	fs->regs.reg[3].loc.offset = (long)&sc->rbx - new_cfa;
	fs->regs.reg[4].how = REG_SAVED_OFFSET;
	fs->regs.reg[4].loc.offset = (long)&sc->rsi - new_cfa;
	fs->regs.reg[5].how = REG_SAVED_OFFSET;
	fs->regs.reg[5].loc.offset = (long)&sc->rdi - new_cfa;
	fs->regs.reg[6].how = REG_SAVED_OFFSET;
	fs->regs.reg[6].loc.offset = (long)&sc->rbp - new_cfa;
	fs->regs.reg[8].how = REG_SAVED_OFFSET;
	fs->regs.reg[8].loc.offset = (long)&sc->r8 - new_cfa;
	fs->regs.reg[9].how = REG_SAVED_OFFSET;
	fs->regs.reg[9].loc.offset = (long)&sc->r9 - new_cfa;
	fs->regs.reg[10].how = REG_SAVED_OFFSET;
	fs->regs.reg[10].loc.offset = (long)&sc->r10 - new_cfa;
	fs->regs.reg[11].how = REG_SAVED_OFFSET;
	fs->regs.reg[11].loc.offset = (long)&sc->r11 - new_cfa;
	fs->regs.reg[12].how = REG_SAVED_OFFSET;
	fs->regs.reg[12].loc.offset = (long)&sc->r12 - new_cfa;
	fs->regs.reg[13].how = REG_SAVED_OFFSET;
	fs->regs.reg[13].loc.offset = (long)&sc->r13 - new_cfa;
	fs->regs.reg[14].how = REG_SAVED_OFFSET;
	fs->regs.reg[14].loc.offset = (long)&sc->r14 - new_cfa;
	fs->regs.reg[15].how = REG_SAVED_OFFSET;
	fs->regs.reg[15].loc.offset = (long)&sc->r15 - new_cfa;
	fs->regs.reg[16].how = REG_SAVED_OFFSET;
	fs->regs.reg[16].loc.offset = (long)&sc->rip - new_cfa;
	fs->retaddr_column = 16;
return _URC_NO_REASON;
Following line:

```c
unsigned char *pc = context->ra;
```

Assigns our controllable return address to the "pc" pointer (program counter). Without any validation following references are done:

```c
if (*(unsigned char *)(pc+0) == 0x48 && *(unsigned long long *)(pc+1) == RT_SIGRETURN_SYSCALL)
```

That’s why SSP by default crashes at this lines of code whenever return address is overwritten by random address. What will happen if we point it to the controllable and valid memory (which of course may be safely referenced)?

```c
struct ucontext *uc_ = context->cfa;
sc = (struct sigcontext *) (void *) &uc_->uc_mcontext;
```

At this point we control signal context:

```c
struct sigcontext *sc;
```

The rest of the code fills in frame state ("_Unwind_FrameState *fs") using our controllable values. After this operation code will return to the main unwinding loop ("_Unwind_Backtrace" function). Next helper function overtake the control via following call:

```c
if ((*trace) (&context, trace_argument) != _URC_NO_REASON)
```

As I described this function before, helper function is responsible for checking if there is any “progress” in stack unwinding by analyzing CFA. It also prevents from the looping around the same frames. What is important it uses following data:

```c
return (_Unwind_Ptr) context->ra; <- _Unwind_GetIP function
return (_Unwind_Ptr) context->cfa; <- _Unwind_GetCFA function
```

Both values are fully controllable. In the end of the unwinding loop, context is updated ("uw_update_context"). At this point new frame is found and parsed (using our fully controllable data):

```c
/* CONTEXT describes the unwind state for a frame, and FS describes the FDE of its caller. Update CONTEXT to refer to the caller as well. Note that the args_size and lsda members are not updated here, but later in uw_frame_state_for. */
static void
```
uw_update_context (struct _Unwind_Context *context, _Unwind_FrameState *fs)
{
    uw_update_context_1 (context, fs);

    /* In general this unwinder doesn't make any distinction between
    undefined and same_value rule. Call-saved registers are
    assumed
    to have same_value rule by default and explicit undefined
    rule is handled like same_value. The only exception is
    DW_CFA_undefined on retaddr_column which is supposed to
    mark outermost frame in DWARF 3. */
    if (fs->regs.reg[DWARF_REG_TO_UNWIND_COLUMN (fs->retaddr_column)].how
        == REG_UNDEFINED)
        /* uw_frame_state_for uses context->ra == 0 check to find
        outermost
        stack frame. */
        context->ra = 0;
    else
        /* Compute the return address now, since the return address
        column
        can change from frame to frame. */
        context->ra = __builtin_extract_return_addr
            (_Unwind_GetPtr (context, fs->retaddr_column));
}

This is just a wrapper to "uw_update_context_1". Before we analyze it, let's quickly look for further
if-else block. We are interested in "else" case which updates return address in the context:

    context->ra = __builtin_extract_return_addr
        (_Unwind_GetPtr (context, fs->retaddr_column));

Quoting the gcc documentation:

"Built-in Function:
void * __builtin_extract_return_addr (void *addr)

The address as returned by __builtin_return_address may have to be
fed through this function to get the actual encoded address. For
example, on the 31-bit S/390 platform the highest bit has to be
masked out, or on SPARC platforms an offset has to be added for
the true next instruction to be executed.

If no fixup is needed, this function simply passes through addr."

What does "_Unwind_GetPtr" do?

static inline void *
_Unwind_GetPtr (struct _Unwind_Context *context, int index)
{ return (void *)(_Unwind_Ptr) _Unwind_GetGR (context, index); }

where:

_Unwind_GetGR (struct _Unwind_Context *context, int index)
{
    int size;
    _Unwind_Context_Reg_Val val;

    #ifdef DWARF_ZERO_REG
    if (index == DWARF_ZERO_REG)
        return 0;
    #endif

    index = DWARF_REG_TO_UNWIND_COLUMN (index);
    gcc_assert (index < (int) sizeof(dwarf_reg_size_table));
    size = dwarf_reg_size_table[index];
    val = context->reg[index];

    if (_Unwind_IsExtendedContext (context) && context->by_value[index])
        return _Unwind_Get_Unwind_Word (val);

    /* This will segfault if the register hasn't been saved. */
    if (size == sizeof(_Unwind_Ptr))
        return * (_Unwind_Ptr *) (_Unwind_Internal_Ptr) val;
    else
    {
        gcc_assert (size == sizeof(_Unwind_Word));
        return * (_Unwind_Word *) (_Unwind_Internal_Ptr) val;
    }
}

In short, this function takes from the context register, value corresponded to the "index" value. It will be value from the return address register in our case.

What does "uw_update_context_1" function do? It's complicated function which plays with CFA. In short function trying to calculate CFA through the saved frame pointer. If frame pointer is not saved (might happen in many architectures or in case of "-fomit-frame-pointer" flag) tracking new CFA is done via analyzing previous one.

After recalculating new CFA, context is updated by the current registers value in that specific frame. In some cases "execute_stack_op" function is executed. It's again complicated function which operates on gcc internal structures. In this case function:

"Decode a DW_OP stack program"

Which is DWARF expression. If every function is finished and new return address is calculated, whole main loop is executed again to analyze newly calculated context for current frame (newly found one). The whole
story starts again. If newly calculated frame have return address pointing somewhere in the unreachable memory, program will crash at read AV (immediate dereference of return address pointer which shouldn’t be trusted).

(Un)Fortunately I was not able to change read AV to any kind of write AV or anything controllable which can give me code execution. Maybe I’m too stupid to play with DWARF algorithm and someone finds a way how to do that. Be aware that we control almost whole context and internal structures, but I was not able to find a way of controlling any metadata in this algorithm (we can look at it as state/stack machine), excluding CFA itself and context which is used for dumping necessary informations (debugging, so memory sections are parsed etc.) and calculating next/new frame...

It’s also common to be killed by "gcc_unreachable" function. It’s called whenever some internal function detects that values in the context which points to the critical data are not as it supposed to be. Similar situation can happen by "gcc_assert" function. Everything need to be perfect aligned and has perfect values if we don’t want to be killed...

In further section of this write-up I’m going to simulate this scenario under debugger (gdb).

OK, this was the case if algorithm didn’t find any FDE. Would be nice to see what might happen in case any FDE was found.

The last scenario (hard one) is in case of calculating FDE object. In this case is even worse and more complicated ;) In theory we have bigger chance of creating write AV / code exec, (un)fortunately I was not able to do that neither. Let’s start...

Function "uw_frame_state_for" instead of calling "x86_64_fallback_frame_state" goes further...

```c
fs->pc = context->bases.func;

cie = get_cie (fde);
insn = extract_cie_info (cie, context, fs);
if (insn == NULL)
    /* CIE contained unknown augmentation. */
    return _URC_FATAL_PHASE1_ERROR;

/* First decode all the insns in the CIE. */
end = (const unsigned char *) next_fde ((const struct dwarf_fde *) cie);
execute_cfa_program (insn, end, context, fs);
```

If FDE was found, CIE object is calculated relative to FDE:

```c
static inline const struct dwarf_cie *
get_cie (const struct dwarf_fde *f)
{
    return (const void *)&f->CIE_delta - f->CIE_delta;
}
"extract_cie_info" function parses current CIE object and extract necessary information which are assigned to the frame state ("fs") structure. Additional this function return pointer to the byte after the augmentation or NULL if undecipherable augmentation was encountered. From this pointer next FDE is calculated to get all possible instructions for the current FDE. Now it's time to execute BIG and complicated function - "execute_cfa_program".

At first, how next FDE is calculated? In very simple way:

```c
static inline const fde *
next_fde (const fde *f)
{
    return (const fde *) ((const char *) f + f->length + sizeof (f->length));
}
```

Let's back to the main problem. What does "execute_cfa_program" do? Quoting internal comments:

/* Decode DWARF 2 call frame information. Takes pointers the instruction sequence to decode, current register information and CIE info, and the PC range to evaluate. */

Further:

/* The comparison with the return address uses < rather than <= because we are only interested in the effects of code before the call; for a noreturn function, the return address may point to unrelated code with a different stack configuration that we are not interested in. We assume that the call itself is unwind info-neutral; if not, or if there are delay instructions that adjust the stack, these must be reflected at the point immediately before the call insn. In signal frames, return address is after last completed instruction, so we add 1 to return address to make the comparison <=. */

Apparently this function "emulates" DWARF instruction and/or expression. It may be seen as a core of DWARF state (stack) machine. In the mean time frame status ("fs") is updated using currently parsed data. If unexpected bytes are parsed, process is killed via "gcc_unreachable" function.

This function is executed twice in the "uw_frame_state_for" function - for current FDE and upper one. Next function is returned to the main loop and this process might happen again (or previously analyzed one). I was not able to force this algorithm (state machine) to execute my code or to do write-AV. Again only Read AV or killing process was achieved.
The main problem is we can’t create own FDE but we can still use existing one by confusing DWARF machine (via controlling return address). Every program has hundreds of existing FDEs. Even if developer didn’t write any Exception Handler (EH), dynamic libraries may have one. Additionally gcc may create some. Following listening shows how many potential FDEs exists in example program and in glibc:

```
[p3@localhost ~]$ readelf -w ./test|grep FDE|sort -u|uniq
00000018 00000014 0000001c FDE cie=00000000 pc=00400540..0040056a
00000048 00000024 0000001c FDE cie=00000030 pc=004004d0..00400540
00000070 0000001c 00000044 FDE cie=00000030 pc=00400630..004006b3
00000090 00000044 00000064 FDE cie=00000030 pc=004006c0..00400725
000000d8 00000014 000000ac FDE cie=00000030 pc=00400730..00400732
[p3@localhost ~]$ readelf -w /lib/libc-2.17.so|grep FDE|sort -u|uniq|wc -l
3665
```

FDE with DWARF expressions (not only DWARF instructions):

```
[p3@localhost ~]$ readelf -w ./test|grep DW_OP
DW_CFA_def_cfa_expression (DW_OP_breg7 (rsp): 8; DW_OP_breg16 (rip): 0; DW_OP_lit15; DW_OP_and; DW_OP_lit11;
DW_OP_ge; DW_OP_lit3; DW_OP shl; DW_OP_plus)
[p3@localhost ~]$ readelf -w /lib/libc-2.17.so|grep DW_OP|wc -l
2140
```

The Lord of the rings and DWARF stories… ;)

At this point I would like to dream a bit... What may happen if we were able to create own FDE? We would be able to create any DWARF instruction and/or expression! In that case we can try to exploit DWARF state (stack) machine itself. Is there any potential code for it? Apparently yes… gcc fixed important bug in DWARF on May 17, 2013. Let’s look for DWARF DW_CFA_register instruction before fix:

```
case DW_CFA_register:
{                  
    _uleb128_t reg2;
    insn_ptr = read_uleb128 (insn_ptr, &reg);
    insn_ptr = read_uleb128 (insn_ptr, &reg2);
    fs->regs.reg[DWARF_REG_TO_UNWIND_COLUMN (reg)].how =
    REG_SAVED_REG;
    fs->regs.reg[DWARF_REG_TO_UNWIND_COLUMN (reg)].loc.reg
    = (_Unwind_Word)reg2;
}  
break;
```

and after:

```
case DW_CFA_register:
{                  
    _uleb128_t reg2;
    insn_ptr = read_uleb128 (insn_ptr, &reg);
```
insn_ptr = read_uleb128 (insn_ptr, &reg2);
reg = DWARF_REG_TO_UNWIND_COLUMN (reg);
if (UNWIND_COLUMN_IN_RANGE (reg))
{
    fs->regs.reg[reg].how = REG_SAVED_REG;
    fs->regs.reg[reg].loc.reg = (_Unwind_Word)reg2;
}
break;

where:

#define UNWIND_COLUMN_IN_RANGE(x) __builtin_expect((x) <= DWARF_FRAME_REGISTERS, 1)

And:

#define DWARF_FRAME_REGISTERS 17

Any code compiled by gcc without following patch, is trivial to exploit – but this is just random ideas.

Btw. Very interesting challenge was introduce in codegate 2014 CTF which required DWARF exploiting as well. Only one team solved this problem – PPP. In their case EH was executed by throwing an exception in SIGSEGV handler. They had primitive to overwrite "frame_hdr_cache_head" pointer which points to the resolved FDEs (EH). Because of that they was able to create own FDE and CIE object, which are parsed in the DWARF exception handling algorithm. If personality routine exist (knowledge based on the controllable CIE object), pointer is extracted, "fs" updated and in the end EH called. I do recommend to read Brian Pak’s write-up on his blog:

http://www.bpak.org/blog/2014/02/codegate-2014-membership-800pt-pwnable-write-up/

In further section of this write-up I’m going to simulate similar scenario under debugger (gdb).

Random ideas...

Not security related...

OK, let's summarize what can be done (with SPP) from the non-security perspective:

1. We can change program’s name (from SSP perspective) via overwriting memory region where pointer to "argv[0]" points to.
2. We can crash Stack Smashing Protector code in many ways:
   a. Via corrupting memory region pointed by "__environ" variable.
   b. Via setting "LIBC_FATAL_STDERR_" to the edge of valid addresses.
   c. Via forcing "alloca()" to fail – e.g. stack exhaustion.
   d. There is one more bug which I’m analyzing more comprehensively at point 4. It may indirectly force SSP to crash. It exists in DWARF stack (state) machine which is responsible for gathering information about the stack trace ("__backtrace()") and prints it.
3. We can slightly control SSP's execution flow. (Un)Fortunately it doesn't have any influence for the main execution (what about security?). Following scenarios are possible:
   a. Force SSP to open "/dev/tty"
   b. Force SSP not to open "/dev/tty" and assign to the "fd" descriptor "STDERR_FILENO" value:

```
#define STDERR_FILENO 2 /* Standard error output. */
```
   c. Crash SSP via 2b. scenario

4. We can crash indirectly SSP via unwinding algorithm (read-AV or we can be killed by "gcc_unreachable" or "gcc_assert" function) – DWARF stack (state) machine:
   a. Simulate FDE object was not found
   b. Simulate FDE object was found.

Somehow security related...

1. We can force SSP to allocate a lot of memory and cause Denial of Service via Resource Exhaustion attack.

This need to be explained a little bit more... We are controlling a following variable:

```
ssize_t total = 0;
```

Which at one point is used as second argument in the "mmap()" function:

```
void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);
```

Someone may realize that "ssize_t" is cast to the "size_t" type. Apparently it doesn't matter here. What is important, how is "(s)size_t" defined? C99 standard from 2007 says:

"7.17 Common definitions <stddef.h>
...

size_t
which is the unsigned integer type of the result of the sizeof operator;"

and reading further:

"The types used for size_t and ptrdiff_t should not have an integer conversion rank greater than that of signed long int unless the implementation supports objects large enough to make this necessary."
Which effectively means for 32 bits "(s)size_t" == "int" but for 64 bits ",(s)size_t" == "long". This type always covers whole process memory address space. Because of that we can’t overflow "total" variable. We may only control one component used for calculation. This component is calculated via "strlen()" function and will never returns need number (around 0xFFFFF000 on 32 bits and around 0xFFFFFFFFFFFF000 for 64 bits).

(Un)Fortunately we are still able to force SSP to dynamically allocate relatively large piece of memory and force it to recopy existing data from the process memory space to the newly allocated buffer. If you do it in very careful way you can try to point to the data which was paged out (swapped out) which force system to execute relatively heavy operation of paging in this data again to the user’s working set. Next because each page in the newly allocated memory will be referenced (copy operation via "memcpy ()" function) system will generate page fault for them and will be forced to make real allocation (make it available in current working set) and recopy physical data from one physical page to another. Additional if the longest consistent chunk of memory you can find, it will be more effective attack.

Of course it’s still controversial if we may assign that scenario to the Resource Exhaustion bucket or not... From my perspective it should be but I can also understand if someone disagree with that.

2. Theoretical Information leak.

If you look closer to the Not security related ideas at point 3b in some very rare situations (impossible in real world?) remote information leak vulnerability may exists. This scenario statically assign to the "fd" descriptor value 2 which by default corresponds to the output stream for errors. Unfortunately you have no guarantee that process didn’t change that. It is possible that application map this descriptor to some opened client’s socket (e.g. via common "dup2 ()" function). This scenario may happened for any application (library?) which emulates pseudoterminals etc. Even more rare situation may happened, if application for some reason closes descriptor number 2 and later tries to open anything (file, device, etc) or tries to create socket, by default this number (2) will be reused (may be done e.g. by some kind of vulnerability which allows you to close what you want). SSP sends the output like application name (which is read from the pointer which may be corrupted), stack trace, etc. exactly to the "fd" descriptor. If you are able to corrupt application name pointer to something you want to leak, it will be send to the descriptor 2 (which could be a socket corresponded to the client’s connection).

Possible theoretical attack scenarios:

a) **Stack cookie Information leak.** If application call any protected function after the stack overflow happened, cookie won’t be overflowed and it will be still save to leak it from the stack. **This scenario allows you to defeat SSP protection in two shots.** First, you force SSP to leak stack cookie. Second shot, you prepare fully working overflow stream (which will include correct cookie value). Of course this scenario will be possible only in "fork ()"-like applications (exclude applications which do "fork ()" + "exec* ()" like OpenSSH or Postfix).
You also need to know stack segment address. If ASLR is enabled, first you can leak stack segment address and next continue original stack cookie information leak attack.

b) **Any kind of information leak.** You can leak whatever you like. It may be useful for ASLR defeating as normal image process leak (shared library base address, if PIE binary, program’s image base address). Leak any kind of the secret from application if won’t be destroyed via this theoretical attack itself.

c) **File corruption.** If descriptor will be assigned (correspond) to any file, SSP’s output will corrupt this file. Another theoretical scenario is when you corrupt program’s name pointer to the data which you fully control, you can corrupt the file with the data which you exactly want. Especially dangerous if critical files are opened (like "passwd / shadow / services", etc.).

**Lazy practice...**

I’m too lazy to test all possible scenarios that’s why I did only few of them...

At first let’s create very simple vulnerable program:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    char buf[100];

    memset(buf,0x0,sizeof(buf));
    if (argv[1])
        strcpy(buf,argv[1]);

    printf("DONE!\n");
    return 0;
}
```

and compile with "-fstack-protector-all" flag:

```
[pi3@localhost ~]$ gcc test.c -o test -g -ggdb -fstack-protector-all
```

```
test.c: In function `main':
test.c:7:4: warning: incompatible implicit declaration of built-in function `memset' [enabled by default]
    memset(buf,0x0,sizeof(buf));
^`
```
test.c:9:7: warning: incompatible implicit declaration of built-in function 'strcpy' [enabled by default]
               strcpy(buf,argv[1]);
^ 
/pi3@localhost ~]$ ./test `perl -e 'print "A"x110'`
DONE!
*** stack smashing detected ***: ./test terminated
======= Backtrace: ========
/lib64/libc.so.6(_fortify_fail+0x37)[0x35c190d6b7]
/lib64/libc.so.6(_fortify_fail+0x0)[0x35c190d680]
/test[0x40075a]
/lib64/libc.so.6(_libc_start_main+0xf5)[0x35c1821b75]
/test[0x4005f9]
======= Memory map: ========
00400000-00401000 r-xp 00000000 fd: 02 262194
/home/pi3/test
00600000-00601000 rw-p 00000000 fd: 02 262194
/home/pi3/test
00601000-00602000 r-xp 00000000 fd: 02 262194
/home/pi3/test
00602000-00603000 r--p 00000000 fd: 02 262194
/home/pi3/test
018e7000-01908000 r-xp 00000000 fd: 02 262194
[heap]
35c1000000-35c1021000 r-xp 00000000 fd:01 1061612
/usr/lib64/ld-2.17.so
35c11220000-35c11221000 r--p 00020000 fd:01 1061612
/usr/lib64/ld-2.17.so
35c12210000-35c12220000 rw-p 00021000 fd:01 1061612
/usr/lib64/ld-2.17.so
35c12220000-35c12230000 rw-p 00000000 00:00 0
35c1320000-35c1321b6000 r-xp 00000000 fd: 02 1061613
/usr/lib64/libc-2.17.so
35c121b6000-35c121bb6000 ---p 001b6000 fd:01 1061613
/usr/lib64/libc-2.17.so
35c121bb6000-35c121ba0000 r--p 001b6000 fd:01 1061613
/usr/lib64/libc-2.17.so
35c121ba0000-35c121bc0000 r-p 001b6000 fd:01 1061613
/usr/lib64/libc-2.17.so
35c121bc0000-35c121d8000 rw-p 00000000 00:00 0
35c14000000-35c40150000 r-xp 00000000 fd:01 1061670
/usr/lib64/libgcc_s-4.8.1-20130603.so.1
35c40150000-35c401f5000 ---p 00150000 fd:01 1061670
/usr/lib64/libgcc_s-4.8.1-20130603.so.1
35c401f5000-35c40214000 r--p 00150000 fd:01 1061670
/usr/lib64/libgcc_s-4.8.1-20130603.so.1
35c40214000-35c40215000 r-xp 00014000 fd:01 1061670
/usr/lib64/libgcc_s-4.8.1-20130603.so.1
35c40215000-35c40216000 rw-p 00015000 fd:01 1061670
/usr/lib64/libgcc_s-4.8.1-20130603.so.1
7f3ab2d94000-7f3ab2d97000 rw-p 00000000 00:00 0
7f3ab2d97000-7f3ab2dab000 rw-p 00000000 00:00 0
7ffe57436000-7ffe57457000 rw-p 00000000 00:00 0
[stack]
7ffe575d4000-7ffe575d6000 r-xp 00000000 00:00 0
[vdso]
As you can see SSP works correctly, detects overflow, appropriate information was gained, and process was killed. Everything was printed to the terminal. As we saw in the SSP/Glibc code, SSP also printed following line:

*** stack smashing detected ***: ./test terminated

Now let's try to cause some bugs in the SSP.

At first, let’s try to check if it is possible to change program’s name - which is a key point for causing theoretical security related bugs.

Normal SSP run:

(gdb) r `perl -e 'print "A"x110'`
Starting program: /home/pi3/test `perl -e 'print "A"x110'`
DONE!
*** stack smashing detected ***: /home/pi3/test terminated

Memory map:

00400000-00401000 r-xp 00000000 00:00 0
/home/pi3/test
00600000-00601000 r--p 00000000 00:00 0
/home/pi3/test
00601000-00602000 rw-p 00001000 00:00 0
/home/pi3/test
00602000-00623000 rw-p 00000000 00:00 0
[heap]
35c100000-35c1021000 r-xp 00000000 fd:01 1061612
/usr/lib64/ld-2.17.so
35c1220000-35c1221000 r--p 00020000 fd:01 1061612
/usr/lib64/ld-2.17.so
35c1221000-35c1222000 rw-p 00021000 fd:01 1061612
/usr/lib64/ld-2.17.so
35c1222000-35c1223000 rw-p 00000000 00:00 0
35c1800000-35c19b6000 r-xp 00000000 fd:01 1061613
/usr/lib64/libc-2.17.so
35c19b6000-35c1bb6000 --p 001b6000 fd:01 1061613
/usr/lib64/libc-2.17.so
35c1bb6000-35c1bb6000 r--p 001b6000 fd:01 1061613
/usr/lib64/libc-2.17.so
Program received signal SIGABRT, Aborted.
0x00000035c1835a19 in __GI_raise (sig=sig@entry=6) at ../nptl/sysdeps/unix/sysv/linux/raise.c:56
 56    return INLINE_SYSCALL (tgkill, 3, pid, selftid, sig);
(gdb) print __libc_argv[0]
$10 = 0x7fffffffe445 "/home/pi3/test"
(gdb)

Program received signal SIGSEGV, Segmentation fault.
__GI_getenv (name=0x35c197bc64 "BC_FATAL_STDERR_",
name@entry=0x35c197bc62 "LIBC_FATAL_STDERR_") at getenv.c:89
 89        if (name_start == ep_start && !strncmp (*ep + 2, name, len)
(gdb) print __libc_argv[0]
$11 = 0x4141414141414141 <Address 0x4141414141414141 out of bounds>
(gdb)

Done. As we expected – it’s possible. Also, stack trace is not printed and program crashed somewhere – we hit one of the described bugs.
Crash 2a:

(gdb) r `perl -e 'print "A"*5000'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/pi3/test `perl -e 'print "A"*5000'`
DONE!

Program received signal SIGSEGV, Segmentation fault.
__GI_getenv (name=0x35c197bc64 "BC_FATAL_STDERR_",
name@entry=0x35c197bc62 "LIBC_FATAL_STDERR_") at getenv.c:89
 89        if (name_start == ep_start && !strncmp (*ep + 2, name,
len)
(gdb) bt
#0  __GI_getenv (name=0x35c197bc64 "BC_FATAL_STDERR_",
name@entry=0x35c197bc62 "LIBC_FATAL_STDERR_") at getenv.c:89
#1  0x00000035c18391c2 in __GI__libc_secure_getenv
(name=name@entry=0x35c197bc62 "LIBC_FATAL_STDERR_") at secure-
getenv.c:30
#2  0x00000035c1875a9a in __libc_message
(do_abort=do_abort@entry=2,
  fmt=fmt@entry=0x35c197d302 "*** %s ***: %s terminated
") at /
../sysdeps/unix/sysv/linux/libc_fatal.c:66
#3  0x00000035c190d6b7 in __GI__fortify_fail
(msg=msg@entry=0x35c197d2ea "stack smashing detected") at for-
tify_fail.c:31
#4  0x00000035c190d680 in __stack_chk_fail () at
stack_chk_fail.c:28
#5  0x000000000004006b1 in main (argc=2, argv=0x7fffffffce58) at
test.c:15
(gdb) list
84    #else
85    uint16_t ep_start = (((unsigned char *) *ep)[0]
86         | (((unsigned char *) *ep)[1] << 8));
87    #endif
88
89    if (name_start == ep_start && !strncmp (*ep + 2, name,
len)
90      && (*ep)[len + 2] == '=')
91      return &(*ep)[len + 3];
92  }
93 }
(gdb) x/i $rip
=> 0x35c183892d <__GI_getenv+173>:  cmp   (%rbx),%r12w
(gdb) i r rbx
rbx 0x41414141414141
4702111234474983745
(gdb) print ep
$12 = (char **) 0x7fffffffce70
(gdb) print *ep
$13 = 0x4141414141414141 <Address 0x4141414141414141 out of bounds>
(gdb)

Crash 2d:

(gdb) r `perl -e 'print "A"x300'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/pi3/test `perl -e 'print "A"x300'`
DONE!
*** stack smashing detected ***: /home/pi3/test terminated

Program received signal SIGSEGV, Segmentation fault.
x86_64_fallback_frame_state (context=0x7fffffffffd3a0,
context=0x7fffffffffd3a0, fs=0x7fffffffffd490) at ./md-unwind-support.h:58
58 if (*(unsigned char *) (pc+0) == 0x48
(gdb) bt
#0 x86_64_fallback_frame_state (context=0x7fffffffffd3a0,
context=0x7fffffffffd3a0, fs=0x7fffffffffd490)
at ./md-unwind-support.h:58
#1 uw_frame_state_for (context=context@entry=0x7fffffffffd3a0,
fs=fs@entry=0x7fffffffffd490)
at ../../../libgcc/unwind-dw2.c:1253
#2 0x00000035c400ff19 in __Unwind_Backtrace (trace=0x35c1909bc0
<backtrace_helper>, trace_argument=0x7fffffffffd650)
at ../../../libgcc/unwind.inc:290
#3 0x00000035c1909d36 in __GI___backtrace_array=0x7fffffffd830, size=size@entry=64)
at ../sysdeps/x86_64/backtrace.c:109
#4 0x00000035c1875d64 in __libc_message
(do_abort=do_abort@entry=2,
_fmt=fmt@entry=0x35c197d302 "**** %s ***: %s terminated\n") at ../sysdeps/unix/sysv/linux/libc_fatal.c:176
#5 0x00000035c190d6b7 in ___GI___fortify_fail
(msg=msg@entry=0x35c197d2ea "stack smashing detected") at forti
fy_fail.c:31
#6 0x00000035c190d60 in __stack_chk_fail () at stack_chk_fail.c:28
#7 0x00000000040006b1 in main (argc=2, argv=0x7fffffffffe0b8) at test.c:15
(gdb) print context->ra
$14 = (void *) 0x4141414141414141
(gdb) x/i $rip
=> 0x35c400f018 <uw_frame_state_for+1080>: cmpb $0x48,(%rcx)
(gdb) i r rcx
rcx 0x4141414141414141 470211123474983745
(gdb) list 50
45 x86_64_fallback_frame_state (struct _Unwind_Context *context,
46      _Unwind_FrameState *fs)
47  {
48    unsigned char *pc = context->ra;
49    struct sigcontext *sc;
50    long new_cfa;
51
52    /* movq $__NR_rt_sigreturn, %rax ; syscall. */
53  #ifdef __LP64__
54  #define RT_SIGRETURN_SYSCALL    0x050f0000000fc0c7ULL
55  #else
56  #define RT_SIGRETURN_SYSCALL    0x050f40000201c0c7ULL
57  #endif
58    if (*(unsigned char *)(pc+0) == 0x48
59        && *(unsigned long long *)(pc+1) ==
60      RT_SIGRETURN_SYSCALL)
61    {
62      struct ucontext *uc_ = context->cfa;
63      /* The void * cast is necessary to avoid an aliasing
warning.

Scenario 3a:

[pi3@localhost ~]$ gdb -q -p 16473
Attaching to process 16473
Reading symbols from /home/pi3/test...done.
Reading symbols from /lib64/libc.so.6...Reading symbols from
/usr/lib/debug/lib64/libc-2.17.so.debug...done.
done.
Loaded symbols for /lib64/libc.so.6
Reading symbols from /lib64/ld-linux-x86-64.so.2...Reading
symbols from /usr/lib/debug/lib64/ld-2.17.so.debug...done.
done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
0x00000035c18e7650 in __read_nocancel () at
../sysdeps/unix/syscall-template.S:81
81  T_PSEUDO (SYSCALL_SYMBOL, SYSCALL_NAME, SYSCALL_NARGS)
(gdb) break libc_fatal.c:66
Breakpoint 1 at 0x35c1875a37: file
../sysdeps/unix/sysv/linux/libc_fatal.c, line 66.
(gdb) c
Continuing.

Breakpoint 1, __libc_message (do_abort=do_abort@entry=2,
fmt=fmt@entry=0x35c197d302 "*** %s ***: %s terminated\n")
at ../sysdeps/unix/sysv/linux/libc_fatal.c:66
66      char *on_2 = __libc_secure getenv
("LIBC_FATAL_STDERR_");
(gdb) list
61      FATAL_PREPARE;
62      #endif
63 /* Open a descriptor for /dev/tty unless the user explicitly 
64 requests errors on standard error. */
65 const char *on_2 = __libc_secure_getenv
   ("LIBC_FATAL_STDERR_");
66 if (on_2 == NULL || *on_2 == '0')
67    fd = open_not_cancel_2 (_PATH_TTY, O_RDWR | O_NOCTTY |
68       O_NDELAY);
69  
70 if (fd == -1)
    (gdb) print on_2
   $1 = <optimized out>
    (gdb) break libc_fatal.c:67
Breakpoint 2 at 0x35c1875a9a: file
   ../sysdeps/unix/sysv/linux/libc_fatal.c, line 67.
    (gdb) c
Continuing.

Breakpoint 2, __libc_message (do_abort=do_abort@entry=2,
   fmt=fmt@entry=0x35c197d302 "*** %s ***: %s terminated\n")
   at ../sysdeps/unix/sysv/linux/libc_fatal.c:67
67 if (on_2 == NULL || *on_2 == '0')
    (gdb) print on_2
   $2 = 0x0
    (gdb) break libc_fatal.c:70
Breakpoint 3 at 0x35c1875abb: file
   ../sysdeps/unix/sysv/linux/libc_fatal.c, line 70.
    (gdb) c
Continuing.

Breakpoint 3, __libc_message (do_abort=do_abort@entry=2,
   fmt=fmt@entry=0x35c197d302 "*** %s ***: %s terminated\n")
   at ../sysdeps/unix/sysv/linux/libc_fatal.c:70
70 if (fd == -1)
   <some inline debug jump>
   <some inline debug jump>
   <some inline debug jump>
    (gdb) ni
68    fd = open_not_cancel_2 (_PATH_TTY, O_RDWR | O_NOCTTY |
    O_NDELAY);
    (gdb) print fd
   $4 = -1
    (gdb) ni
70 if (fd == -1)
    (gdb) print fd
   $5 = 3
    (gdb)

and double check in the list of opened file descriptors for this process:
Scenario 3b:

[gpi3@localhost ~] $ ls -al /proc/16473/fd
total 0
dr-x-------.  2 pi3 pi3  0 Sep 29 11:02  .
dr-xr-xr-x.  9 pi3 pi3  0 Sep 29 11:02 ..
lrwx-------.  1 pi3 pi3  64 Sep 29 11:06 0 -> /dev/pts/3
lrwx-------.  1 pi3 pi3  64 Sep 29 11:06 1 -> /dev/pts/3
lrwx-------.  1 pi3 pi3  64 Sep 29 11:06 2 -> /dev/pts/3
lrwx-------.  1 pi3 pi3  64 Sep 29 11:06 3 -> /dev/tty

[gpi3@localhost ~] $ gdb -q -p 16531
Attaching to process 16531
Reading symbols from /home/pi3/test...done.
Reading symbols from /lib64/libc.so.6...Reading symbols from /usr/lib/debug/lib64/libc-2.17.so.debug...done.
Loaded symbols for /lib64/libc.so.6
Reading symbols from /lib64/ld-linux-x86-64.so.2...Reading symbols from /usr/lib/debug/lib64/ld-2.17.so.debug...done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
0x00000035c18e7650 in __read_nocancel () at ../sysdeps/unix/syscall-template.S:81
81  T_PSEUDO (SYSCALL_SYMBOL, SYSCALL_NAME, SYSCALL_NARGS)
   (gdb) break libc_fatal.c:66
Breakpoint 1 at 0x35c1875a37: file
   ../sysdeps/unix/sysv/linux/libc_fatal.c, line 66.
   (gdb) break libc_fatal.c:67
Breakpoint 2 at 0x35c1875a9a: file
   ../sysdeps/unix/sysv/linux/libc_fatal.c, line 67.
   (gdb) c
Continuing.

Breakpoint 1, __libc_message (do_abort=do_abort@entry=2,
fmt=fmt@entry=0x35c197d302 "*** %s ***: %s terminated\n")
   at ../sysdeps/unix/sysv/linux/libc_fatal.c:66
66   const char *on_2 = __libc_secure_getenv
("LIBC_FATAL_STDERR_");
   (gdb) break libc_fatal.c:70
Breakpoint 3 at 0x35c1875abb: file
   ../sysdeps/unix/sysv/linux/libc_fatal.c, line 70.
   (gdb) print on_2
$1 = <optimized out>
   (gdb) c
Continuing.

Breakpoint 2, __libc_message (do_abort=do_abort@entry=2,
fmt=fmt@entry=0x35c197d302 "*** %s ***: %s terminated\n")
and double check:

```
[pi3@localhost ~]$ ls -al /proc/16531/fd
total 0
dr-x-------. 2 pi3 pi3  0 Sep 29 11:11 .
dr-xr-xr-x. 9 pi3 pi3  0 Sep 29 11:11 ..
lrwx-------. 1 pi3 pi3 64 Sep 29 11:14 0 -> /dev/pts/3
lrwx-------. 1 pi3 pi3 64 Sep 29 11:14 1 -> /dev/pts/3
lrwx-------. 1 pi3 pi3 64 Sep 29 11:11 2 -> /dev/pts/3
[pi3@localhost ~]$
```

Scenario 4a:

At the beginning let's just prove that simple return address overflow might lead to read-AV crash. First overflow cookie without touching return address:

```
[pi3@localhost ~]$ gdb -q ./test
Reading symbols from /home/pi3/test...(no debugging symbols found)...done.
(gdb) r `perl -e 'print "A"x120'`
Starting program: /home/pi3/test `perl -e 'print "A"x120'`
DONE!
*** stack smashing detected ***: /home/pi3/test terminated
======== Backtrace: ========
/lib64/libc.so.6(__fortify_fail+0x37)[0x35c190d6b7]
/lib64/libc.so.6(__fortify_fail+0x0)[0x35c190d680]
/home/pi3/test[0x4006b1]
/lib64/libc.so.6(__libc_start_main+0x80)[0x35c1821b00]
/home/pi3/test[0x400569]
======== Memory map: ========
00400000-00401000 r-xp 00000000 fd:02 262194
/home/pi3/test
00600000-00601000 r--p 00000000 fd:02 262194
/home/pi3/test
00601000-00602000 rw-p 00001000 fd:02 262194
/home/pi3/test
00602000-00623000 rw-p 00000000 00:00 0
[heap]
35c1000000-35c1021000 r-xp 00000000 fd:01 1061612
/usr/lib64/ld-2.17.so
35c1220000-35c1221000 r--p 00020000 fd:01 1061612
/usr/lib64/ld-2.17.so
35c1221000-35c1222000 rw-p 00021000 fd:01 1061612
/usr/lib64/ld-2.17.so
```
Program received signal SIGABRT, Aborted.
0x00000035c1835a19 in __GI_raise (sig=sig@entry=6) at
  ../nptl/sysdeps/unix/sysv/linux/raise.c:56
  return INLINE_SYSCALL (tgkill, 3, pid, selftid, sig);
(gdb) bt
#0 0x00000035c1835a19 in __GI_raise (sig=sig@entry=6) at
  ../nptl/sysdeps/unix/sysv/linux/raise.c:56
#1 0x00000035c1837128 in __GI_abort () at abort.c:90
#2 0x00000035c1875d47 in __libc_message
  (do_abort=do_abort@entry=2,
   fmt=fmt@entry=0x35c197d302 "*** %s ***: %s terminated\n") at
  ../sysdeps/unix/sysv/linux/libc_fatal.c:196
#3 0x00000035c190d6b7 in __GI__fortify_fail
  (msg=msg@entry=0x35c197d2ea "stack smashing detected") at
  fortify_fail.c:31
#4 0x00000035c190d680 in __stack_chk_fail () at
  stack_chk_fail.c:28
#5 0x00000000004006b1 in main ()
(gdb) list libc_fatal.c:196
191  close_not_cancel_no_status (fd2);
192  }
193 }
194
195 /* Terminate the process. */
196 abort();
Overwrite return address:

```
(gdb) r `perl -e 'print "A"x124'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/pi3/test `perl -e 'print "A"x124'`
DONE!
*** stack smashing detected ***: /home/pi3/test terminated
```

Program received signal SIGSEGV, Segmentation fault.
x86_64_fallback_frame_state (context=0x7fffffffd420, context=0x7fffffffd420, fs=0x7fffffffd510) at ./md-unwind-support.h:58
58 if (*(unsigned char *)(pc+0) == 0x48
(gdb) bt
#0 x86_64_fallback_frame_state (context=0x7fffffffd420, context=0x7fffffffd420, fs=0x7fffffffd510)
at ./md-unwind-support.h:58
#1 uw_frame_state_for (context=context@entry=0x7fffffffd420, fs=fs@entry=0x7fffffffd510)
at ../../../libgcc/unwind-dw2.c:1253
#2 0x00000035c400ff19 in __Unwind_Backtrace (trace=0x35c1909bc0 <backtrace_helper>, trace_argument=0x7fffffffd6d0)     
at ../../../libgcc/unwind.inc:290
#3 0x00000035c1909d36 in __GI___backtrace (array=array@entry=0x7fffffffd510, size=size@entry=64)
at ../sysdeps/x86_64/backtrace.c:109
#4 0x00000035c1875d64 in __libc_message (do_abort=do_abort@entry=2,
  fmt=fmt@entry=0x35c197d302 "*** %s ***: %s terminated\n") at ../sysdeps/unix/sysv/linux/libc_fatal.c:176
#5 0x00000035c190d6b7 in __GI___fortify_fail (msg=msg@entry=0x35c197d2ea "stack smashing detected") at fortify_fail.c:31
#6 0x00000035c190d680 in __stack_chk_fail () at stack_chk_fail.c:28
#7 0x00000035c190d6b1 in main ()
(gdb) print pc
$2 = (unsigned char *) 0x41414141 <Address 0x41414141 out of bounds>
(gdb)
```

As you can see, instead of killing the process, SIGSEGV was received. The crash happened exactly where we predict in our previous analyze. Let’s now simulate that we fully control memory where return address point to (signal frame):
(gdb) r `perl -e 'print "A"x300'`

The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/pi3/test `perl -e 'print "A"x300'`

DONE!
*** stack smashing detected ***: /home/pi3/test terminated

Breakpoint 9, uw_frame_state_for
(context=context@entry=0x7fffffffd370,
fs=fs@entry=0x7fffffffd1c0)
at ../../../libgcc/unwind-dw2.c:1233
1233 }
(gdb) c
...<many times>
...<let's pad the memory>
...<let's pad the memory>
...<many times>
if (fde == NULL) {
    #ifdef MD_FALLBACK_FRAME_STATE_FOR
    /* Couldn't find frame unwind info for this function. Try a
     * target-specific fallback mechanism. This will necessarily
     * not provide a personality routine or LSDA. */
    return MD_FALLBACK_FRAME_STATE_FOR (context, fs);
    #else
    return _URC_END_OF_STACK;
    #endif
}
(gdb) b 1247
Breakpoint 20 at 0x35c400ec8c: file ../../../libgcc/unwind-dw2.c, line 1247.
(gdb) c
Continuing.

Breakpoint 20, uw_frame_state_for
(context=context@entry=0x7fffffffdf370,
fs=fs@entry=0x7fffffffdf460)
    at ../../../libgcc/unwind-dw2.c:1247
1247   if (fde == NULL)
(gdb) del 20
(gdb) list
1232   uw_frame_state_for (struct _Unwind_Context *context,
               _Unwind_FrameState *fs)
(gdb)
(gdb) x86_64_fallback_frame_state (context=0x7fffffffdf370, context=0x7fffffffdf370, fs=0x7fffffffdf460) at ./md-unwind-support.h:68
68     return _URC_END_OF_STACK;
(gdb)
58     if (*((unsigned char *)(pc+0)) == 0x48
57     {
56         #ifdef __LP64__
55         #define RT_SIGRETURN_SYSCALL 0x050f0000000fc0c7ULL
54         #else
53         #define RT_SIGRETURN_SYSCALL 0x050f40000201c0c7ULL
52     #endif
51     if (*((unsigned char *)(pc+0)) == 0x48
50         && *((unsigned long long *)(pc+1)) == RT_SIGRETURN_SYSCALL)
49     {
48         struct ucontext *uc_ = context->cfa;
47         /* The void * cast is necessary to avoid an aliasing warning. */
46         sc = (struct sigcontext *) (void *) &uc_->uc_mcontext;
45     }
44     else
43     return _URC_END_OF_STACK;
42
41     new_cfa = sc->rsp;
40     fs->regs.cfa_how = CFA_REG_OFFSET;
41     /* Register 7 is rsp */
(gdb) x/20i $rip
=> 0x35c400f018 <uw_frame_state_for+1080>: cmpb $0x48,(%rcx)
0x35c400f01b <uw_frame_state_for+1083>: jne 0x35c400eded <uw_frame_state_for+525>
0x35c400f021 <uw_frame_state_for+1089>: movabs $0x50f000000fc0c7,%rsi
0x35c400f02b <uw_frame_state_for+1099>: cmp %rsi,0x1(%rcx)
0x35c400f02f <uw_frame_state_for+1103>: jne 0x35c400eded <uw_frame_state_for+525>
0x35c400f035 <uw_frame_state_for+1109>: mov 0xa0(%rdx),%rax
0x35c400f03c <uw_frame_state_for+1116>: lea 0x90(%rdx),%rsi
0x35c400f043 <uw_frame_state_for+1123>: movl $0x1,0x140(%r12)
0x35c400f04f <uw_frame_state_for+1135>: movq $0x7,0x130(%r12)
0x35c400f05b <uw_frame_state_for+1147>: movl $0x1,0x8(%r12)
<we need to change the memory layout to pass the checks>

...
struct ucontext *uc_ = context->cfa;
/* The void * cast is necessary to avoid an aliasing
warning. The aliasing warning is correct, but should not be
a problem (gdb)
because it does not alias anything. */
sc = (struct sigcontext *) (void *) &uc_>
>uc_mcontext;
}
else
    return _URC_END_OF_STACK;

new_cfa = sc->rsp;
fs->regs.cfa_how = CFA_REG_OFFSET;
/* Register 7 is rsp */
fs->regs.cfa_reg = 7;
(gdb) si
new_cfa = sc->rsp;
(gdb) print sc
$163 = (struct sigcontext *) 0x7fffffffdfd8
(gdb) print/x *sc
$165 = {r8 = 0x4141414141414141, r9 = 0x4141414141414141, r10 = 0x4141414141414141, r11 = 0x4141414141414141, r12 = 0x4141414141414141, r13 = 0x4141414141414141, r14 = 0x4141414141414141, rdi = 0x4141414141414141, rsi = 0x4141414141414141, 
        rbp = 0x4141414141414141, 
        rdx = 0x4141414141414141, 
        rip = 0x7f0041414141, 
        eflags = 0x0, cs = 0x569, gs = 0x40, fs = 0x0, __pad0 = 0x0, err = 0x7fffffffde078, trapno = 0x1c, 
        oldmask = 0x2, cr2 = 0x7fffffffde376, {fpstate = 0x7fffffffde376, __fpstate_word = 0x7fffffffde376}, __reserved1 = {0x0, 
        0x7fffffffde4a3, 0x7fffffffde4ae, 0x7fffffffde4c0, 
        0x7fffffffde4df, 0x7fffffffde514, 0x7fffffffde52b, 0x7fffffffde53b}}
(gdb) si
fs->regs.reg[0].loc.offset = (long)&sc->rax - new_cfa;
(gdb)
fs->regs.cfa_how = CFA_REG_OFFSET;
(gdb)
fs->regs.cfa_reg = 7;
(gdb)
fs->regs.reg[0].how = REG_SAVED_OFFSET;
(gdb)
fs->regs.reg[1].how = REG_SAVED_OFFSET;
(gdb)
fs->regs.reg[2].how = REG_SAVED_OFFSET;
(gdb)
fs->regs.cfa_offset = new_cfa - (long) context->cfa;
(gdb)
fs->regs.reg[0].loc.offset = (long)&sc->rax - new_cfa;
fs->regs.reg[3].how = REG_SAVED_OFFSET;

fs->regs.cfa_offset = new_cfa - (long) context->cfa;

fs->regs.reg[0].loc.offset = (long)&sc->rax - new_cfa;

fs->regs.reg[1].loc.offset = (long)&sc->rdx - new_cfa;

fs->regs.cfa_offset = new_cfa - (long) context->cfa;

fs->regs.reg[0].loc.offset = (long)&sc->rax - new_cfa;

fs->regs.reg[4].how = REG_SAVED_OFFSET;

fs->regs.reg[1].loc.offset = (long)&sc->rdx - new_cfa;

fs->regs.reg[5].how = REG_SAVED_OFFSET;

fs->regs.reg[6].how = REG_SAVED_OFFSET;

fs->regs.reg[8].loc.offset = (long)&sc->r8 - new_cfa;

fs->regs.reg[1].loc.offset = (long)&sc->rdx - new_cfa;

fs->regs.reg[2].loc.offset = (long)&sc->rcx - new_cfa;

fs->regs.reg[8].loc.offset = (long)&sc->r8 - new_cfa;

fs->regs.reg[9].loc.offset = (long)&sc->r9 - new_cfa;

fs->regs.reg[8].how = REG_SAVED_OFFSET;

fs->regs.reg[2].loc.offset = (long)&sc->rcx - new_cfa;

fs->regs.reg[9].how = REG_SAVED_OFFSET;

fs->regs.reg[10].how = REG_SAVED_OFFSET;

fs->regs.reg[9].loc.offset = (long)&sc->r9 - new_cfa;

fs->regs.reg[2].loc.offset = (long)&sc->rcx - new_cfa;

fs->regs.reg[3].loc.offset = (long)&sc->rbx - new_cfa;

fs->regs.reg[9].loc.offset = (long)&sc->r9 - new_cfa;

fs->regs.reg[10].loc.offset = (long)&sc->r10 - new_cfa;

fs->regs.reg[11].how = REG_SAVED_OFFSET;

fs->regs.reg[3].loc.offset = (long)&sc->rbx - new_cfa;
(gdb)
99  fs->regs.reg[12].how = REG_SAVED_OFFSET;
(gdb)
101 fs->regs.reg[13].how = REG_SAVED_OFFSET;
(gdb)
96  fs->regs.reg[10].loc.offset = (long)&sc->r10 - new_cfa;
(gdb)
84  fs->regs.reg[3].loc.offset = (long)&sc->rbx - new_cfa;
(gdb)
86  fs->regs.reg[4].loc.offset = (long)&sc->rsi - new_cfa;
(gdb)
96  fs->regs.reg[10].loc.offset = (long)&sc->r10 - new_cfa;
(gdb)
98  fs->regs.reg[11].loc.offset = (long)&sc->r11 - new_cfa;
(gdb)
103 fs->regs.reg[14].how = REG_SAVED_OFFSET;
(gdb)
86  fs->regs.reg[4].loc.offset = (long)&sc->rsi - new_cfa;
(gdb)
105 fs->regs.reg[15].how = REG_SAVED_OFFSET;
(gdb)
98  fs->regs.reg[11].loc.offset = (long)&sc->r11 - new_cfa;
(gdb)
86  fs->regs.reg[4].loc.offset = (long)&sc->rsi - new_cfa;
(gdb)
88  fs->regs.reg[5].loc.offset = (long)&sc->rdi - new_cfa;
(gdb)
98  fs->regs.reg[11].loc.offset = (long)&sc->r11 - new_cfa;
(gdb)
100 fs->regs.reg[12].loc.offset = (long)&sc->r12 - new_cfa;
(gdb)
88  fs->regs.reg[5].loc.offset = (long)&sc->rdi - new_cfa;
(gdb)
100 fs->regs.reg[12].loc.offset = (long)&sc->r12 - new_cfa;
(gdb)
88  fs->regs.reg[5].loc.offset = (long)&sc->rdi - new_cfa;
(gdb)
90  fs->regs.reg[6].loc.offset = (long)&sc->rbp - new_cfa;
(gdb)
100 fs->regs.reg[12].loc.offset = (long)&sc->r12 - new_cfa;
(gdb)
102 fs->regs.reg[13].loc.offset = (long)&sc->r13 - new_cfa;
(gdb)
90  fs->regs.reg[6].loc.offset = (long)&sc->rbp - new_cfa;
(gdb)
102 fs->regs.reg[13].loc.offset = (long)&sc->r13 - new_cfa;
(gdb)
90  fs->regs.reg[6].loc.offset = (long)&sc->rbp - new_cfa;
(gdb)
102 fs->regs.reg[13].loc.offset = (long)&sc->r13 - new_cfa;
(gdb)
104 fs->regs.reg[14].loc.offset = (long)&sc->r14 - new_cfa;
(gdb)
0x00000035c400f1be 104 fs->regs.reg[14].loc.offset =
(long)&sc->r14 - new_cfa;
(gdb)
0x00000035c400f1c1 104 fs->regs.reg[14].loc.offset =
(long)&sc->r14 - new_cfa;
(gdb)
106  fs->regs.reg[15].loc.offset = (long)&sc->r15 - new_cfa;
(gdb)
108  fs->regs.reg[16].loc.offset = (long)&sc->rip - new_cfa;
(gdb)
0x00000035c400f1d4 108 fs->regs.reg[16].loc.offset =
(long)&sc->rip - new_cfa;
(gdb)
106  fs->regs.reg[15].loc.offset = (long)&sc->r15 - new_cfa;
(gdb)
111  return _URC_NO_REASON;
(gdb)
106  fs->regs.reg[15].loc.offset = (long)&sc->r15 - new_cfa;
(gdb)
107  fs->regs.reg[16].how = REG_SAVED_OFFSET;
(gdb)
108  fs->regs.reg[16].loc.offset = (long)&sc->rip - new_cfa;
(gdb)
109  fs->retaddr_column = 16;
(gdb)
110  fs->signal_frame = 1;
(gdb)
0x00000035c400f20d 110 fs->signal_frame = 1;
(gdb)
uw_frame_state_for (context=cont
ext@entry=0x7fffffffd370,
fs=fs@entry=0x7fffffffd460) at ../../../libgcc/unwind-dw2.c:1296
1296 }
(gdb)
0x00000035c400edf1 1296 }
(gdb) list
0x00000035c400edf2 1296 }
(gdb) si
(gdb) 0x00000035c400edf7  1296 }
(gdb) 0x00000035c400edf9  1296 }
(gdb) 0x00000035c400edfb  1296 }
(gdb) si 
_Unwind_Backtrace (trace=0x35c1909bc0 <backtrace_helper>,
  trace_argument=0x7fffffffffd620) at ../../../libgcc/unwind.inc:291
291  if (code != _URC_NO_REASON && code !=
  _URC_END_OF_STACK)
(gdb) 290     code = uw_frame_state_for (&context, &fs);
(gdb) 291     if (code != _URC_NO_REASON && code !=
  _URC_END_OF_STACK)
(gdb) 0x00000035c400ff21  291     if (code != _URC_NO_REASON &&
    code != _URC_END_OF_STACK)
(gdb) 0x00000035c400ff23  291     if (code != _URC_NO_REASON &&
    code != _URC_END_OF_STACK)
(gdb) 295     if (::trace) (&context, trace_argument) !=
   _URC_NO_REASON)
(gdb) 0x00000035c400ff6f  295     if (::trace) (&context, trace_argument) !=
   _URC_NO_REASON)
(gdb) n
299     if (code == _URC_END_OF_STACK)
(gdb) print code
$167 = _URC_NO_REASON
(gdb) n
303     uw_update_context (&context, &fs);
(gdb) print fs->regs.cfa_how
$168 = CFA_REG_OFFSET
(gdb) p/x context->cfa
$169 = 0x7fffffffffd6b0
(gdb) si
0x00000035c400ff6f  303     uw_update_context (&context,
    &fs);
(gdb) 0x00000035c400ff9f  303     uw_update_context (&context,
    &fs);
(gdb) uw_update_context (context=context@entry=0x7fffffffffd370,
  fs=fs@entry=0x7fffffffffd460) at ../../../libgcc/unwind-dw2.c:1505
1505 {
(gdb) 0x00000035c400eb51  1505 {
(gdb) 0x00000035c400eb54  1505 {

52
(gdb)
0x00000035c400eb55  1505 {
(gdb)
0x00000035c400eb58  1505 {
(gdb)
1506  uw_update_context_1 (context, fs);
(gdb)
uw_update_context_1 (context=context@entry=0x7fffffffddd70,
fs=fs@entry=0x7fffffffddd460) at ../../../libgcc/unwind-dw2.c:1382
1382 {
(gdb)
1383  struct __Unwind_Context orig_context = *context;
(gdb) n
1382 {
(gdb)
1383  struct __Unwind_Context orig_context = *context;
(gdb)
1382 {
(gdb)
1383  struct __Unwind_Context orig_context = *context;
(gdb)
1405  if (!_Unwind_GetGRPtr (&orig_context, 
    __builtin_dwarf_sp_column ()))
(gdb)
1406  _Unwind_SetSpColumn (&orig_context, context->cfa, 
    &tmp_sp);
(gdb)
1407  _Unwind_SetGRPtr (context, __builtin_dwarf_sp_column (), 
    NULL);
(gdb)
1411  switch (fs->regs.cfa_how)
(gdb)
1407  _Unwind_SetGRPtr (context, __builtin_dwarf_sp_column (), 
    NULL);
(gdb)
1411  switch (fs->regs.cfa_how)
(gdb)
1414  cfa = _Unwind_GetPtr (&orig_context, fs->
    regs.cfa_reg);
(gdb)
1415  cfa += fs->regs.cfa_offset;
(gdb)
1416  break;
(gdb)
1436  switch (fs->regs.reg[i].how)
(gdb)
1432  context->cfa = cfa;
(gdb)
1467  _Unwind_SetGRPtr (context, i, (void *) val);
(gdb)
1436  switch (fs->regs.reg[i].how)
(gdb)
for (i = 0; i < DWARF_FRAME_REGISTERS + 1; ++i)
switch (fs->regs.reg[i].how)
    _Unwind_SetGRPtr (context, i,
(gdb)
for (i = 0; i < DWARF_FRAME_REGISTERS + 1; ++i)
switch (fs->regs.reg[i].how)
    _Unwind_SetGRPtr (context, i,
(gdb)
for (i = 0; i < DWARF_FRAME_REGISTERS + 1; ++i)
switch (fs->regs.reg[i].how)
    _Unwind_SetGRPtr (context, i,
(gdb)
for (i = 0; i < DWARF_FRAME_REGISTERS + 1; ++i)
switch (fs->regs.reg[i].how)
    _Unwind_SetGRPtr (context, i,
(gdb)
for (i = 0; i < DWARF_FRAME_REGISTERS + 1; ++i)
switch (fs->regs.reg[i].how)
    _Unwind_SetGRPtr (context, i,
(gdb)
for (i = 0; i < DWARF_FRAME_REGISTERS + 1; ++i)
switch (fs->regs.reg[i].how)
    _Unwind_SetGRPtr (context, i,
1443    _Unwind_SetGRPtr (context, i,
(gdb)
1435     (gdb)
1436     switch (fs->regs.reg[i].how)
(gdb)
1433     _Unwind_SetGRPtr (context, i,
(gdb)
1435     for (i = 0; i < DWARF_FRAME_REGISTERS + 1; ++i)
(gdb)
1436     switch (fs->regs.reg[i].how)
(gdb)
1443     _Unwind_SetGRPtr (context, i,
(gdb)
1435     for (i = 0; i < DWARF_FRAME_REGISTERS + 1; ++i)
(gdb)
1436     switch (fs->regs.reg[i].how)
(gdb)
1443     _Unwind_SetGRPtr (context, i,
(gdb)
1435     for (i = 0; i < DWARF_FRAME_REGISTERS + 1; ++i)
(gdb)
1436     switch (fs->regs.reg[i].how)
(gdb)
1443     _Unwind_SetGRPtr (context, i,
(gdb)
1435     for (i = 0; i < DWARF_FRAME_REGISTERS + 1; ++i)
(gdb)
1436     switch (fs->regs.reg[i].how)
(gdb)
1443     _Unwind_SetGRPtr (context, i,
(gdb)
1435     for (i = 0; i < DWARF_FRAME_REGISTERS + 1; ++i)
(gdb)
1436     switch (fs->regs.reg[i].how)
(gdb)
1443     _Unwind_SetGRPtr (context, i,
(gdb)
1435     for (i = 0; i < DWARF_FRAME_REGISTERS + 1; ++i)
(gdb)
1436     switch (fs->regs.reg[i].how)
(gdb)
1443     _Unwind_SetGRPtr (context, i,
(gdb)
1435     for (i = 0; i < DWARF_FRAME_REGISTERS + 1; ++i)
(gdb)
1436     switch (fs->regs.reg[i].how)
(gdb)
1443     _Unwind_SetGRPtr (context, i,
(gdb)
1435     for (i = 0; i < DWARF_FRAME_REGISTERS + 1; ++i)
(gdb)
1436     switch (fs->regs.reg[i].how)
(gdb)
1443     _Unwind_SetGRPtr (context, i,
(gdb)
1435     for (i = 0; i < DWARF_FRAME_REGISTERS + 1; ++i)
(gdb)
1436     switch (fs->regs.reg[i].how)
(gdb)
1435     for (i = 0; i < DWARF_FRAME_REGISTERS + 1; ++i)
(gdb)
1491    _Unwind_SetSignalFrame (context, fs->signal_frame);
1496 } 
(gdb) uw_update_context (context=context@entry=0x7fffffffd370, 
fs=fs@entry=0x7fffffffd460) at ../../../libgcc/unwind-dw2.c:1514
1514 if (fs->regs.reg[DWARF_REG_TO_UNWIND_COLUMN (fs-> 
retaddr_column)].how 
(gdb) 1523 (_Unwind_GetPtr (context, fs->retaddr_column)); 
(gdb) 1522 context->ra = __builtin_extract_return_addr 
(gdb) n 1524 
(gdb) print context->ra
$170 = (void *) 0x7f0041414141 <- Newly calculated return address
(gdb) list 1519 else 1520 /* Compute the return address now, since the return 
address column 1521 can change from frame to frame. */ 
1522 context->ra = __builtin_extract_return_addr 
1523 (_Unwind_GetPtr (context, fs->retaddr_column)); 
1524 }
1525
1526 static void
1527 uw_advance_context (struct _Unwind_Context *context, 
_Uwind_FrameState *fs)
1528 {
(gdb) x/20i $rip => 0x35c400ebba <uw_update_context+106>: add $0x8,%rsp 
0x35c400ebbe <uw_update_context+110>: pop %rbx 
0x35c400ebbf <uw_update_context+111>: pop %rbp 
0x35c400ecb0 <uw_update_context+112>: retq 
0x35c400ecb1 <uw_update_context+113>: nopl 0x0(%rax) 
0x35c400ecb8 <uw_update_context+120>: movq 
$0x0,0x98(%rbx) 
0x35c400ecbd <uw_update_context+131>: add $0x8,%rsp 
0x35c400eced <uw_update_context+135>: pop %rbx 
0x35c400eced <uw_update_context+136>: pop %rbp 
0x35c400eced9 <uw_update_context+137>: retq 
0x35c400ebe0 <uw_frame_state_for>: nopw 0x0(%rax,%rax,1) 
0x35c400ebe0 <uw_frame_state_for>: push %r15 
0x35c400ebe2 <uw_frame_state_for+2>: mov $0x180,%edx 
0x35c400ebe7 <uw_frame_state_for+7>: push %r14 
0x35c400ebe9 <uw_frame_state_for+9>: push %r13 
0x35c400ebef <uw_frame_state_for+11>: mov %rdi,%r13 
0x35c400ebef <uw_frame_state_for+14>: mov %rsi,%rdi 
0x35c400ebf1 <uw_frame_state_for+17>: push %r12 
0x35c400ebf3 <uw_frame_state_for+19>: mov %rsi,%r12 
0x35c400ebf6 <uw_frame_state_for+22>: push %rbp 
(gdb) i r rcx 
rcx 0x7f0041414141 139639071523137 
(gdb) p/x *fs
The latest values in the context and frame state variables corresponds to the relative offsets from the 0x4141414141414141 values. Next iteration of the main loop eventually cause the read-AV because currently calculated return address is pointing to the unreachable memory. We returned to the starting point.
Scenario 4b – let’s try to check if DWARF instructions are emulated if we fully control FDE:

```plaintext
(gdb) r `perl -e 'print "A"x300'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/pi3/test `perl -e 'print "A"x300'`
DONE!
*** stack smashing detected ***: /home/pi3/test terminated

Breakpoint 9, uw_frame_state_for
   (context=context@entry=0x7fffffff6fd370,
    fs=fs@entry=0x7fffffff6fd1c0)
   at ../../../libgcc/unwind-dw2.c:1233
1233 {
(gdb) c
Continuing.
...
<keep going until we hit what we want...>
...
(gdb) source pi3-test  <- prepare the memory and do padding
Breakpoint 18 at 0x35c400ec8c: file ../../../libgcc/unwind-dw2.c, line 1247.

Breakpoint 18, uw_frame_state_for
   (context=context@entry=0x7fffffff6fd370,
    fs=fs@entry=0x7fffffff6fd460)
   at ../../../libgcc/unwind-dw2.c:1247
1247 if (fde == NULL)
1245 fde = _Unwind_Find_FDE (context->ra +
   _Unwind_IsSignalFrame (context) - 1,
1247 if (fde == NULL)
1259 fs->pc = context->bases.func;
get_cie (f=<optimized out>) at ../../../libgcc/unwind-dw2-fde.h:157
157 return (const void *)&f->CIE_delta - f->CIE_delta;
uw_frame_state_for (context=context@entry=0x7fffffff6fd370,
    fs=fs@entry=0x7fffffff6fd460) at ../../../libgcc/unwind-dw2.c:1259
1259 fs->pc = context->bases.func;
get_cie (f=0x7fffffff6de101) at ../../../libgcc/unwind-dw2-fde.h:157
157 return (const void *)&f->CIE_delta - f->CIE_delta;
0x00000035c400ecaf 157 return (const void *)&f->CIE_delta -
   f->CIE_delta;
(gdb) si
extract_cie_info (fs=0x7fffffff6fd460, context=0x7fffffff6fd370,
cie=0x7fffffff6bebc9fc4) at ../../../libgcc/unwind-dw2.c:415
415 const unsigned char *aug = cie->augmentation;
(gdb) set cie = context->ra
(gdb) print *cie
$138 = {length = 1094795585, CIE id = 1094795585, version = 65
   'A', augmentation = 0x7fffffff6de10a 'A' <repeats 200 times>...
```

58
(gdb) set cie->version = 0x1

<fix the memory to avoid killing by gcc_unreachable() or gcc_assert()>

(gdb) print *cie
$139 = {length = 1094795585, CIE_id = 1094795585, version = 1 '001', augmentation = 0x7ffffffde10a 'A' <repeats 200 times>...}
(gdb) si 416   const unsigned char *p = aug + strlen ((const char *)aug) + 1;
(gdb) 0x00000035c400ecb9 416   const unsigned char *p = aug + strlen ((const char *)aug) + 1;
(gdb) n 423   if (aug[0] == 'e' && aug[1] == 'h')
(gdb) si 416   const unsigned char *p = aug + strlen ((const char *)aug) + 1;
(gdb) 423   if (aug[0] == 'e' && aug[1] == 'h')
(gdb) 433   if (__builtin_expect (cie->version >= 4, 0))
(gdb) x/20i $rip
=> 0x35c400ecce <uw_frame_state_for+238>: cmpb $0x3,0x8(%r14)
0x35c400ecd3 <uw_frame_state_for+243>: ja 0x35c400f592
<uw_frame_state_for+2482>
0x35c400ecd9 <uw_frame_state_for+249>: xor %esi,%esi
0x35c400ecd3 <uw_frame_state_for+251>: xor %ecx,%ecx
0x35c400ecdb <uw_frame_state_for+253>: nopl (%rax)
0x35c400ece0 <uw_frame_state_for+256>: add $0x1,%rdi
0x35c400ece4 <uw_frame_state_for+260>: movzbl -0x1(%rdi),%edx
0x35c400ece8 <uw_frame_state_for+264>: mov %rdx,%rax
0x35c400eceb <uw_frame_state_for+267>: and $0x7f,%eax
0x35c400ecee <uw_frame_state_for+270>: shl %cl,%rax
0x35c400ecf1 <uw_frame_state_for+273>: add $0x7,%ecx
0x35c400ecf4 <uw_frame_state_for+276>: or %rax,%rsi
0x35c400ecf7 <uw_frame_state_for+279>: test %dl,%dl
0x35c400ecf9 <uw_frame_state_for+281>: js 0x35c400ece0
<uw_frame_state_for+256>
0x35c400ecfb <uw_frame_state_for+283>: mov %rsi,0x160(%r12)
0x35c400ed03 <uw_frame_state_for+291>: lea 0x40(%rsp),%rsi
0x35c400ed08 <uw_frame_state_for+296>: callq 0x35c400d090 <read_sleb128>
0x35c400ed0d <uw_frame_state_for+301>: mov %rax,%rbx
0x35c400ed10 <uw_frame_state_for+304>: mov 0x40(%rsp),%rax
0x35c400ed15 <uw_frame_state_for+309>: xor %esi,%esi
(gdb) x/x $r14+0x8
0x7fffffffde109: 0x01
(gdb) si
0x00000035c400ecd3 433 if (__builtin_expect (cie->version 
>= 4, 0))
(gdb)
x86_64_fallback_frame_state (context=<optimized out>, 
context=<optimized out>, fs=<optimized out>) 
at ./md-unwind-support.h:68
68 return _URC_END_OF_STACK;
(gdb) x/20i $rip
=> 0x35c400ecd9 <uw_frame_state_for+249>: xor %esi,%esi
0x35c400ecdb <uw_frame_state_for+251>: xor %ecx,%ecx
0x35c400ecd3 <uw_frame_state_for+253>: nopl (%rax) 
0x35c400eec0 <uw_frame_state_for+256>: add $0x1,%rdi
0x35c400ee4 <uw_frame_state_for+260>: movzbl -
0x1(%rdi),%edx
0x35c400ece8 <uw_frame_state_for+264>: mov %rdx,%rax
0x35c400eceb <uw_frame_state_for+267>: and $0x7f,%eax
0x35c400eceee <uw_frame_state_for+270>: shl %cl,%rax
0x35c400ecf1 <uw_frame_state_for+273>: add $0x7,%ecx
0x35c400ecf4 <uw_frame_state_for+276>: or %rax,%rsi
0x35c400ecf7 <uw_frame_state_for+279>: test %dl,%dl
0x35c400ecf9 <uw_frame_state_for+281>: js 0x35c400ee0
<uw_frame_state_for+256>
0x35c400ecfb <uw_frame_state_for+283>: mov
%rsi,0x160(%r12)
0x35c400ed03 <uw_frame_state_for+291>: lea
0x40(%rsp),%rsi
0x35c400ed08 <uw_frame_state_for+296>: callq 0x35c400d090
<read_sleb128>
0x35c400ed0d <uw_frame_state_for+301>: mov %rax,%rbx
0x35c400ed10 <uw_frame_state_for+304>: mov
0x40(%rsp),%rax
0x35c400ed15 <uw_frame_state_for+309>: xor %esi,%esi
0x35c400ed17 <uw_frame_state_for+311>: xor %ecx,%ecx
0x35c400ed19 <uw_frame_state_for+313>: mov
%rax,0x158(%r12)
(gdb) si
0x00000035c400ecd3 68 return _URC_END_OF_STACK;
(gdb) 0x00000035c400ecd6 68 return _URC_END_OF_STACK;
(gdb) x/20i $rip
=> 0x35c400ecd6 <uw_frame_state_for+253>: nopl (%rax)
0x35c400ee0 <uw_frame_state_for+256>: add $0x1,%rdi
0x35c400ee4 <uw_frame_state_for+260>: movzbl -
0x1(%rdi),%edx
0x35c400ee8 <uw_frame_state_for+264>: mov %rdx,%rax
0x35c400eeb <uw_frame_state_for+267>: and $0x7f,%eax
0x35c400eece <uw_frame_state_for+270>: shl %cl,%rax
0x35c400ecf1 <uw_frame_state_for+273>: add $0x7,%ecx
0x35c400ecf4 <uw_frame_state_for+276>: or %rax,%rsi
0x35c400ecf7 <uw_frame_state_for+279>:    test  %dl,%dl
0x35c400ecf9 <uw_frame_state_for+281>:    js   0x35c400ece0
<uw_frame_state_for+256>
  0x35c400ecfb <uw_frame_state_for+283>:    mov  %rsi,0x160(%r12)
  0x35c400ed03 <uw_frame_state_for+291>:    lea  0x1340(%r12),%rsi
  0x40(%rsp),%rsi
  0x35c400ed08 <uw_frame_state_for+296>:    callq  0x35c400d090
<read_sleb128>
  0x35c400ed0d <uw_frame_state_for+301>:    mov   %rax,%rbx
  0x35c400ed10 <uw_frame_state_for+304>:    mov   0x40(%rsp),%rax
  0x35c400ed15 <uw_frame_state_for+309>:    xor   %esi,%esi
  0x35c400ed17 <uw_frame_state_for+311>:    xor   %ecx,%ecx
  0x35c400ed19 <uw_frame_state_for+313>:    mov   %rax,0x158(%r12)
  0x35c400ed21 <uw_frame_state_for+321>:    cmpb  $0x1,0x8(%r14)
  0x35c400ed26 <uw_frame_state_for+326>:    je    0x35c400f270
<uw_frame_state_for+1680>
(gdb) si
read_uleb128 (val=<optimized out>, p=0x7ffffffdf0a5 "") at
  ../../../libgcc/unwind-pe.h:140
  byte = *p++;
(gdb)
0x00000035c400ece4  140  byte = *p++;
(gdb)
141  result |= ((__uleb128_t)byte & 0x7f) << shift;
(gdb)
0x00000035c400eceb  141  result |= ((__uleb128_t)byte & 0x7f) << shift;
(gdb)
0x00000035c400eceee 141  result |= ((__uleb128_t)byte & 0x7f) << shift;
(gdb)
142  shift += 7;
(gdb)
141  result |= ((__uleb128_t)byte & 0x7f) << shift;
(gdb)
144  while (byte & 0x80);
(gdb)
0x00000035c400ecf9  144  while (byte & 0x80);
(gdb)
extract_cie_info (fs=0x7fffffffdfd460, context=0x7fffffffdfd370, cie=0x7fffffffde101) at
  ../../../libgcc/unwind-dw2.c:442
442  fs->code_align = (_Unwind_Word)utmp;
(gdb) print utmp
$140 = 0
(gdb) si
443  p = read_sleb128 (p, &stmp);
(gdb) x/20i $rip
=> 0x35c400ed03 <uw_frame_state_for+291>:    lea  0x40(%rsp),%rsi
61
0x35c400ed08 <uw_frame_state_for+296>: callq 0x35c400d090

<read_sleb128>
0x35c400ed0d <uw_frame_state_for+301>: mov %rax,%rbx
0x35c400ed10 <uw_frame_state_for+304>: mov 0x40(%rsp),%rax
0x35c400ed15 <uw_frame_state_for+309>: xor %esi,%esi
0x35c400ed17 <uw_frame_state_for+311>: xor %ecx,%ecx
0x35c400ed19 <uw_frame_state_for+313>: mov %rax,0x158(%r12)
0x35c400ed21 <uw_frame_state_for+321>: cmpb $0x1,0x8(%r14)
0x35c400ed26 <uw_frame_state_for+326>: je 0x35c400f270

<uw_frame_state_for+1680>
0x35c400ed2c <uw_frame_state_for+332>: nopl 0x0(%rax)
0x35c400ed30 <uw_frame_state_for+336>: add $0x1,%rbx
0x35c400ed34 <uw_frame_state_for+340>: movzbl -0x1(%rbx),%edx
0x35c400ed38 <uw_frame_state_for+344>: mov %rdx,%rax
0x35c400ed3b <uw_frame_state_for+347>: and $0x7f,%eax
0x35c400ed41 <uw_frame_state_for+353>: add $0x7,%ecx
0x35c400ed44 <uw_frame_state_for+356>: or %rax,%rsi
0x35c400ed47 <uw_frame_state_for+359>: test %dl,%dl
0x35c400ed49 <uw_frame_state_for+361>: js 0x35c400ed30

<uw_frame_state_for+336>
0x35c400ed4b <uw_frame_state_for+363>: mov 0x1(%rbx),%edx
0x35c400ed38 <uw_frame_state_for+344>: mov %rdx,%rax
0x35c400ed3b <uw_frame_state_for+347>: and $0x7f,%eax
0x35c400ed41 <uw_frame_state_for+353>: add $0x7,%ecx
0x35c400ed44 <uw_frame_state_for+356>: or %rax,%rsi
0x35c400ed47 <uw_frame_state_for+359>: test %dl,%dl
0x35c400ed49 <uw_frame_state_for+361>: js 0x35c400ed30

(gdb) si 0x00000035c400ed08 443 p = read_sleb128 (p, &stmp);
(gdb) read_sleb128 (p=0x7ffffffdf0a6 "", val=val@entry=0x7fffffffd320) at ../../../libgcc/unwind-pe.h:154
154 {
(gdb) 159     result = 0;
(gdb) 155     unsigned int shift = 0;
(gdb) 0x00000035c400d098 155 unsigned int shift = 0;
(gdb) 162     byte = *p++;
(gdb) 0x00000035c400d0a4 162 byte = *p++;
(gdb) 163     result |= ((__uleb128_t)byte & 0x7f) << shift;
(gdb) 0x00000035c400d0ab 163 result |= ((__uleb128_t)byte & 0x7f) << shift;
(gdb) 0x00000035c400d0ae 163 result |= ((__uleb128_t)byte & 0x7f) << shift;
(gdb) 164     shift += 7;
result |= ((_uleb128_t)byte & 0x7f) << shift;

while (byte & 0x80);

if (shift < 8 * sizeof(result) && (byte & 0x40) != 0)
    *val = (_sleb128_t) result;

fs->data_sleb128 (p, &stmp);

fs->data_align = (_Unwind_Sword)stmp;

fs->retaddr_column = *p++;

fs->lsda_encoding = DW_EH_PE_omit;

if (*aug == 'z')
    const unsigned char *ret = NULL;
(gdb) if (*aug == 'z')
(gdb) 0x00000035c400ed65 457 if (*aug == 'z')
(gdb) read_encoded_value_with_base (val=<optimized out>, p=<optimized out>, base=<optimized out>, encoding=<optimized out>)
 at ../../../libgcc/unwind-pe.h:225
225 p = read_sleb128 (p, &tmp);
(gdb) 0x00000035c400ed70 225 p = read_sleb128 (p, &tmp);
(gdb) 207 switch (encoding & 0x0f)
225 p = read_sleb128 (p, &tmp);
(gdb) 0x00000035c400ed80 225 p = read_sleb128 (p, &tmp);
(gdb) extract_cie_info (fs=0x7fffffffd460, context=0x7fffffffd370, cie=0x7ffffffde101) at ../../../libgcc/unwind-dw2.c:467
467 while (*aug != '\0')
(gdb) 0x00000035c400edab 467 while (*aug != '\0')
(gdb) 470 if (aug[0] == 'L')
(gdb) 0x00000035c400edb3 470 if (aug[0] == 'L')
(gdb) 477 else if (aug[0] == 'R')
(gdb) 0x00000035c400ed8a 477 else if (aug[0] == 'R')
(gdb) 484 else if (aug[0] == 'P')
(gdb) 0x00000035c400ed8e 484 else if (aug[0] == 'P')
(gdb) 494 else if (aug[0] == 'S')
(gdb) 0x00000035c400ed92 494 else if (aug[0] == 'S')
(gdb) 494 else if (aug[0] == 'S')
(gdb) uw_frame_state_for (context=context@entry=0x7fffffffd370, fs=fs@entry=0x7fffffffd460) at ../../../../libgcc/unwind-dw2.c:1263
1263 if (insn == NULL)
(gdb) 0x00000035c400f2b6 1263 if (insn == NULL)
(gdb) list
1258
1259 fs->pc = context->bases.func;
1260
1261 cie = get_cie (fde);
insn = extract_cie_info (cie, context, fs);
if (insn == NULL)
/* CIE contained unknown augmentation. */
return _URC_FATAL_PHASE1_ERROR;
/* First decode all the insns in the CIE. */
(gdb) x/20i $rip
=> 0x35c400f2b6 <uw_frame_state_for+1750>: jne 0x35c400eeac
<uw_frame_state_for+716>
 0x35c400f2bc <uw_frame_state_for+1756>: add $0x58,%rsp
 0x35c400f2c0 <uw_frame_state_for+1760>: mov $0x3,%eax
 0x35c400f2c5 <uw_frame_state_for+1765>: pop %rbx
 0x35c400f2c6 <uw_frame_state_for+1766>: pop %rbp
 0x35c400f2c7 <uw_frame_state_for+1767>: pop %r12
 0x35c400f2c9 <uw_frame_state_for+1769>: pop %r13
 0x35c400f2cb <uw_frame_state_for+1771>: pop %r14
 0x35c400f2cd <uw_frame_state_for+1773>: pop %r15
 0x35c400f2cf <uw_frame_state_for+1775>: retq
0x35c400f2d0 <uw_frame_state_for+1776>: lea 0x1(%rsi),%rdi
0x35c400f2d4 <uw_frame_state_for+1780>: movb $0x0,(%rsi)
0x35c400f2d7 <uw_frame_state_for+1783>: mov $0x7f,%dl
0x35c400f2d9 <uw_frame_state_for+1785>: test $0x2,%dl
0x35c400f2dd <uw_frame_state_for+1789>: je 0x35c400ec10
<uw_frame_state_for+48>
 0x35c400f2e3 <uw_frame_state_for+1795>: nop
0x0(%rax,%rax,1)
 0x35c400f2e8 <uw_frame_state_for+1800>: xor %ecx,%ecx
 0x35c400f2ea <uw_frame_state_for+1802>: add $0x2,%rdi
 0x35c400f2ee <uw_frame_state_for+1806>: sub $0x2,%edx
 0x35c400f2f1 <uw_frame_state_for+1809>: mov %cx,-0x2(%rdi)
(gdb) print context->ra
$142 = (void *) 0x7ffffffdde101
(gdb) set $rdi = context->ra
(gdb) si
uw_frame_state_for (context=context@entry=0x7fffffffd370,
fs=fs@entry=0x7fffffffd460) at ../../../libgcc/unwind-dw2.c:1263
1263 if (insn == NULL)
(gdb) x/20i $rip
=> 0x35c400f2b3 <uw_frame_state_for+1747>: test %rdi,%rdi

0x35c400f2b6 <uw_frame_state_for+1750>: jne 0x35c400eeac
<uw_frame_state_for+716>
 0x35c400f2bc <uw_frame_state_for+1756>: add $0x58,%rsp
 0x35c400f2c0 <uw_frame_state_for+1760>: mov $0x3,%eax
 0x35c400f2c5 <uw_frame_state_for+1765>: pop %rbx
 0x35c400f2c6 <uw_frame_state_for+1766>: pop %rbp
 0x35c400f2c7 <uw_frame_state_for+1767>: pop %r12
 0x35c400f2c9 <uw_frame_state_for+1769>: pop %r13
 0x35c400f2cb <uw_frame_state_for+1771>: pop %r14
 0x35c400f2cd <uw_frame_state_for+1773>: pop %r15
 0x35c400f2cf <uw_frame_state_for+1775>: retq
 0x35c400f2d0 <uw_frame_state_for+1776>: lea 0x1(%rsi),%rdi
 0x35c400f2d4 <uw_frame_state_for+1780>: movb $0x0,(%rsi)
0x35c400f2d7 <uw_frame_state_for+178>:  mov $0x7f,%dl
0x35c400f2d9 <uw_frame_state_for+1785>:  test $0x2,%dl
0x35c400f2dd <uw_frame_state_for+1789>:  je 0x35c400ec10

0x35c400f2e3 <uw_frame_state_for+1795>:  nopl
0x0(%rax,%rax,1)
0x35c400f2e8 <uw_frame_state_for+1800>:  xor %ecx,%ecx
0x35c400f2ea <uw_frame_state_for+1802>:  add $0x2,%rdi
0x35c400f2ee <uw_frame_state_for+1806>:  sub $0x2,%edx

(gdb) i r rdi r9
di            0x7ffffffde101 140737488216321
r9             0x7ffffffde101 140737488216321

(gdb) si
0x00000035c400f2b6 1263 if (insn == NULL)
(gdb) next_fde (f=0x7ffffffde101) at ../../../libgcc/unwind-dw2-fde.h:163
163  return (const fde *) ((const char *) f + f->length +
sizeof (f->length));

(gdb) si
uw_frame_state_for (context=context@entry=0x7fffffffd370,
fs=fs@entry=0x7fffffffd460) at ../../../libgcc/unwind-dw2.c:1269
1269  execute_cfa_program (insn, end, context, fs);

(gdb) print context
$144 = (struct _Unwind_Context *) 0x7fffffffd370
(gdb) print *context
$145 = {reg = {0x0, 0x0, 0x0, 0x0, 0x7fffffffd8f, 0x0, 0x0,
0x7fffffffdec0, 0x7fffffffdec8, 0x7fffffffded0, 0x7fffffffded8,
0x7fffffffdec8, 0x0}, cfa = 0x7fffffffdd8f, ra = 0x7ffffffde101,
lsda = 0x0, bases = {tbase = 0x0, dbase = 0x0, func = 0x400630
<main>}, flags = 4611686018427387904, version = 0,
args_size = 0, by_value = '\000' <repeats 17 times>}
(gdb) si
0x00000035c400eb2 1269  execute_cfa_program (insn, end,
context, fs);
(gdb) next_fde (f=0x7ffffffde101) at ../../../libgcc/unwind-dw2-fde.h:163
163  return (const fde *) ((const char *) f + f->length +
sizeof (f->length));

(gdb) list
158 }
159
160 static inline const fde *
161 next_fde (const fde *f)
162 {
163  return (const fde *) ((const char *) f + f->length +
sizeof (f->length));
164 }
165
extern const fde * _Unwind_Find_FDE (void *, struct
dwarf_eh_bases *);

(gdb) si
uw_frame_state_for (context=context@entry=0x7fffffffd370,
fs=fs@entry=0x7fffffffd460) at ../../../libgcc/unwind-dw2.c:1269
1269 execute_cfa_program (insn, end, context, fs);
(gdb)
execute_cfa_program (insn_ptr=0x7fffffffd101 "AAAAAAAA\001", 'A'
<repeats 191 times>,
insn_end=0x8000413f2246 <Address 0x8000413f2246 out of
bounds>, context=context@entry=0x7fffffffd370,
fs=fs@entry=0x7fffffffd460) at ../../../libgcc/unwind-
dw2.c:942
942 {
(gdb) si
0x00000035c400d431 942 {
(gdb) si
0x00000035c400d434 942 {
(gdb)
0x00000035c400d436 942 {
(gdb)
0x00000035c400d439 942 {
(gdb)
0x00000035c400d43b 942 {
(gdb)
0x00000035c400d43d 942 {
(gdb)
0x00000035c400d440 942 {
(gdb)
0x00000035c400d442 942 {
(gdb)
0x00000035c400d443 942 {
(gdb)
0x00000035c400d446 942 {
(gdb) 957 while (insn_ptr < insn_end
(gdb) print insn_ptr
$146 = (const unsigned char *) 0x7fffffffd101 "AAAAAAAA\001", 'A'
<repeats 191 times>...
(gdb) print insn_end
$147 = (const unsigned char *) 0x8000413f2246 <Address
0x8000413f2246 out of bounds>
(gdb) print insn_end - insn_ptr
$148 = 1094795589
(gdb) si
946 fs->regs.prev = NULL;
(gdb) 957 while (insn_ptr < insn_end
(gdb) _Unwind_IsSignalFrame (context=<optimized out>) at
../../../libgcc/unwind-dw2.c:202
return (context->flags & SIGNAL_FRAME_BIT) ? 1 : 0;

(gdb) list
197 read_8s (const void *p) { const union unaligned *up = p;
return up->s8; }
198 ^L
199 static inline _Unwind_Word
200 _Unwind_IsSignalFrame (struct _Unwind_Context *context)
201 {
202 return (context->flags & SIGNAL_FRAME_BIT) ? 1 : 0;
203 }
204
205 static inline void
206 _Unwind_SetSignalFrame (struct _Unwind_Context *context, int val)
207 {
execute_cfa_program (insn_ptr=0x7ffffffde101 "AAAAAAAA\001", 'A'
<repeats 191 times>...,
insn_end=0x8000413f2246 <Address 0x8000413f2246 out of bounds>,
context=0x7fffffffe370, fs=fs@entry=0x7fffffffe460) at ../../../libgcc/unwind-
dw2.c:958
958 && fs->pc < context->ra + _Unwind_IsSignalFrame
959 (context))
960 while (insn_ptr < insn_end
961 && fs->pc < context->ra + _Unwind_IsSignalFrame
962 (context))
963 { unsigned char insn = *insn_ptr++;
964 _uleb128_t reg, utmp;
965 _sleb128_t offset, stmp;
966 (gdb) print fs->pc
$149 = (void *) 0x400630 <main>
(gdb) print context->ra
$150 = (void *) 0x7fffffffe101
(gdb) si
0x00000035c400d46c 958 && fs->pc < context->ra + _Unwind_IsSignalFrame
959 (context))
960 (gdb) x/20i $rip
=> 0x35c400d46c <execute_cfa_program+60>: mov %rdx,%r14
0x35c400d46f <execute_cfa_program+63>: shr $0x3f,%rax
0x35c400d473 <execute_cfa_program+67>: add
0x98(%rdx),%rax
0x35c400d47a <execute_cfa_program+74>: cmp %rax,%rcx

0x35c400d47d <execute_cfa_program+77>: jae 0x35c400d50b
<execute_cfa_program+219>
    0x35c400d483 <execute_cfa_program+83>: lea -
0x38(%rbp),%rax
    0x35c400d487 <execute_cfa_program+87>: lea
0x5162(%rip),%rdx          # 0x35c40125f0
    0x35c400d48e <execute_cfa_program+94>: xor %r8d,%r8d
    0x35c400d491 <execute_cfa_program+97>: mov %rax,-
0x48(%rbp)
    0x35c400d495 <execute_cfa_program+101>: nopl (%rax)
    0x35c400d498 <execute_cfa_program+104>: movzbl (%rbx),%eax
    0x35c400d49b <execute_cfa_program+107>: lea 0x1(%rbx),%r12
    0x35c400d49f <execute_cfa_program+111>: mov %eax,%esi
    0x35c400d4af <execute_cfa_program+113>: and
$0xffffffffc0,%esi
    0x35c400d4a4 <execute_cfa_program+116>: cmp $0x40,%sil
    0x35c400d4a8 <execute_cfa_program+120>: je 0x35c400d4d0
<execute_cfa_program+160>
    0x35c400d4aa <execute_cfa_program+122>: cmp $0x80,%sil
    0x35c400d4ae <execute_cfa_program+126>: je 0x35c400d520
<execute_cfa_program+240>
    0x35c400d4b0 <execute_cfa_program+128>: cmp $0xc0,%sil
    0x35c400d4b4 <execute_cfa_program+132>: je 0x35c400d578
<execute_cfa_program+328>
(gdb) si
_Unwind_IsSignalFrame (context=0x7fffffffd370) at
../..//libgcc/unwind-dw2.c:202
202    return (context->flags & SIGNAL_FRAME_BIT) ? 1 : 0;
(gdb)
execute_cfa_program (insn_ptr=0x7fffffffde101 "AAAAAAAA\001", 'A'
<repeats 191 times>..., insn_end=0x8000413f2246 <Address 0x8000413f2246 out of
bounds>, context=context@entry=0x7fffffffde101, 
    fs=fs@entry=0x7fffffffde460) at ../..//libgcc/unwind-
dw2.c:958
958
(gdb) si
0x00000035c400d47a 958  && fs->pc < context->ra + _Unwind_IsSignalFrame
(context))
(gdb) si
0x00000035c400d47a 958  && fs->pc < context->ra + _Unwind_IsSignalFrame (context))
(gdb)
0x00000035c400d47d 958  && fs->pc < context->ra + _Unwind_IsSignalFrame (context))
(gdb)
1169 insn_ptr = read_sleb128 (insn_ptr, &stmp);
(gdb) 985 else switch (insn)
(gdb) print insn
$151 = <optimized out>
(gdb) b *0x35c400d495
Breakpoint 19 at 0x35c400d495: file ../..//libgcc/unwind-dw2.c, line 1169. 69
(gdb) c
Continuing.

Breakpoint 19, 0x00000035c400d495 in execute_cfa_program (insn_ptr=0x7fffffffde101 "AAAAAAAA\001", 'A' <repeats 191 times>...,
  insn_end=0x8000413f2246 <Address 0x8000413f2246 out of bounds>, context=context@entry=0x7fffffffde370,
  fs=fs@entry=0x7fffffffde460) at ../../../libgcc/unwind-dw2.c:1169
1169     insn_ptr = read_sleb128 (insn_ptr, &stmp);
(gdb) x/20i $rip-10
  0x35c400d48b <execute_cfa_program+91>: push %rcx
  0x35c400d48c <execute_cfa_program+92>: add %al,(%rax)
  0x35c400d48e <execute_cfa_program+94>: xor %r8d,%r8d
  0x35c400d491 <execute_cfa_program+97>: mov %rax,-
0x48(%rbp)
=> 0x35c400d495 <execute_cfa_program+101>: nopl (%rax)
  0x35c400d498 <execute_cfa_program+104>: movzbl (%rbx),%eax
  0x35c400d49b <execute_cfa_program+107>: lea 0x1(%rbx),%r12
  0x35c400d49f <execute_cfa_program+111>: mov %eax,%esi
  0x35c400d4a1 <execute_cfa_program+113>: and $0xffffffffc0,%esi
  0x35c400d4a4 <execute_cfa_program+116>: cmp $0x40,%sil
  0x35c400d4a8 <execute_cfa_program+120>: je 0x35c400d4d0
<execute_cfa_program+160>
  0x35c400d4aa <execute_cfa_program+122>: cmp $0x80,%sil
  0x35c400d4ae <execute_cfa_program+126>: je 0x35c400d520
<execute_cfa_program+240>
  0x35c400d4b0 <execute_cfa_program+128>: cmp $0xc0,%sil
  0x35c400d4b4 <execute_cfa_program+132>: je 0x35c400d578
<execute_cfa_program+328>
  0x35c400d4ba <execute_cfa_program+138>: cmp $0x2f,%al
  0x35c400d4bc <execute_cfa_program+140>: ja 0x35c400d5a9
<execute_cfa_program+377>
  0x35c400d4c2 <execute_cfa_program+146>: movslq (%rdx,%rax,4),%rax
  0x35c400d4c6 <execute_cfa_program+150>: add %rdx,%rax
  0x35c400d4c9 <execute_cfa_program+153>: jmpq *%rax
(gdb) x/x $rbx
0x7fffffffde101: 0x41
(gdb) set *(0x7fffffffde101) = 0x1
(gdb) x/8x 0x7fffffffde101
0x7fffffffde101: 0x01 0x00 0x00 0x00 0x41 0x41 0x41 0x41 0x41
(gdb) set *(0x7fffffffde101) = 0x80 <-emulate DWARF instruction>
...
<fix the memory to avoid killing by gcc_unreachable() or gcc_assert()>
...
(gdb) si
960     unsigned char insn = *insn_ptr++;
(gdb)
unsigned char insn = (gdb)
if ((insn & 0xc0) == DW_CFA_advance_loc) (gdb)

if ((insn & 0xc0) == DW_CFA_advance_loc) (gdb)
else if ((insn & 0xc0) == DW_CFA_offset) (gdb)
reg = insn & 0x3f;
reg = insn & 0x3f;
read_uleb128 (val=<optimized out>, p=<optimized out>) at ../../../libgcc/unwind-pe.h:137
result = 0;
execute_cfa_program (insn_ptr=0x7ffffffde102 "",
insn_end=0x8000413f2246 <Address 0x8000413f2246 out of bounds>,
context=context@entry=0x7fffffffdf370,
fs=fs@entry=0x7fffffffdf460) at ../../../libgcc/unwind-dw2.c:968
reg = insn & 0x3f;
(gdb) x/20i $rip
=> 0x35c400d528 <execute_cfa_program+248>: and $0x3f,%edi
0x35c400d52b <execute_cfa_program+251>: xor %ecx,%ecx
0x35c400d52d <execute_cfa_program+253>: nopl (%rax)
0x35c400d530 <execute_cfa_program+256>: add $0x1,%rbx
0x35c400d534 <execute_cfa_program+260>: movzbl -
0x1(%rbx),%r9d
0x35c400d539 <execute_cfa_program+265>: mov %r9,%rax
0x35c400d53c <execute_cfa_program+268>: and $0x7f,%eax
0x35c400d53f <execute_cfa_program+271>: shl %cl,%rax
0x35c400d542 <execute_cfa_program+274>: add $0x7,%ecx
0x35c400d545 <execute_cfa_program+277>: or %rax,%rsi
0x35c400d548 <execute_cfa_program+280>: test %r9b,%r9b
0x35c400d54b <execute_cfa_program+283>: js 0x35c400d530
<execute_cfa_program+256>
0x35c400d54d <execute_cfa_program+285>: imul
0x158(%r13),%rsi
0x35c400d555 <execute_cfa_program+293>: cmp $0x11,%rdi
0x35c400d559 <execute_cfa_program+297>: ja 0x35c400d4e8
<execute_cfa_program+184>
0x35c400d55b <execute_cfa_program+299>: shl $0x4,%rdi
0x35c400d55f <execute_cfa_program+303>: add %r13,%rdi
0x35c400d562 <execute_cfa_program+306>: movl $0x1,0x8(%rdi)
0x35c400d569 <execute_cfa_program+313>: mov %rsi,(%rdi)
0x35c400d56c <execute_cfa_program+316>: jmpq 0x35c400d4e8

(gdb) si
read_uleb128 (val=<synthetic pointer>, p=0x7ffffffde102 "") at
../../../libgcc/unwind-pe.h:133
133 unsigned int shift = 0;
(gdb)
140 byte = *p++;
(gdb)
0x00000035c400d534 140 byte = *p++;
(gdb)
141 result |= ((_uleb128_t)byte & 0x7f) << shift;
(gdb)
0x00000035c400d53c 141 result |= ((_uleb128_t)byte &
0x7f) << shift;
(gdb)
0x00000035c400d53f 141 result |= ((_uleb128_t)byte &
0x7f) << shift;
(gdb)
142 shift += 7;
(gdb)
141 result |= ((_uleb128_t)byte & 0x7f) << shift;
(gdb)
144 while (byte & 0x80);
(gdb)
0x00000035c400d54b 144 while (byte & 0x80);
(gdb)
execute_cfa_program (insn_ptr=0x7ffffffde103 "",
insn_end=0x8000413f2246 <Address 0x8000413f2246 out of bounds>,
context=context@entry=0x7fffffffd370,
fs=fs@entry=0x7fffffffd460) at ../../../libgcc/unwind-dw2.c:970
970 offset = (_Unwind_Sword) utmp * fs->data_align;
(gdb)
972 if (UNWIND_COLUMN_IN_RANGE (reg))
(gdb)
0x00000035c400d559 972 if (UNWIND_COLUMN_IN_RANGE
(reg))
(gdb)
0x00000035c400d55b 972 if (UNWIND_COLUMN_IN_RANGE
(reg))
(gdb)
0x00000035c400d55f 972 if (UNWIND_COLUMN_IN_RANGE
(reg))
(gdb)
974 fs->regs.reg[reg].how = REG_SAVED_OFFSET;
(gdb)
975 fs->regs.reg[reg].loc.offset = offset;
(gdb) print fs
$153 = (_Unwind_FrameState *) 0x7fffffffdf460
(gdb) print *fs
$154 = {regs = {reg = {loc = {reg = 0, offset = 0, exp = 0x0},
how = REG_SAVED_OFFSET}, {loc = {reg = 0, offset = 0,
   exp = 0x0}, how = REG_UNSAVED} <repeats 17 times>},
prev = 0x0, cfa_offset = 0, cfa_reg = 0, cfa_exp = 0x0,
   cfa_how = CFA_UNSET}, pc = 0x400630 <main>, personality =
0x0, data_align = 0, code_align = 0, retaddr_column = 0,
fde_encoding = 0 '\000', lsda_encoding = 255 '\377', saw_z = 0
'\000', signal_frame = 0 '\000', eh_ptr = 0x0}
(gdb) si
0x00000035c400d56c 975 fs-
>regs.reg[reg].loc.offset = offset;
(gdb) print offset
$155 = 0
(gdb) si
957 while (insn_ptr < insn_end
(gdb)
0x00000035c400d4eb 957 while (insn_ptr < insn_end
(gdb)
_Unwind_IsSignalFrame (context=0x7fffffffdf370) at
../../../../../libgcc/unwind-dw2.c:202
202 return (context->flags & SIGNAL_FRAME_BIT) ? 1 : 0;
(gdb)
execute_cfa_program (insn_ptr=0x7fffffffde103 "",
insn_end=0x8000413f2246 <Address 0x8000413f2246 out of bounds>,
   context=context@entry=0x7fffffffdf370,
   fs=fs@entry=0x7fffffffdf460) at ../../../../../libgcc/unwind-dw2.c:958
958 & & fs->pc < context->ra + _Unwind_IsSignalFrame
   (context))
(gdb)
_Unwind_IsSignalFrame (context=0x7fffffffdf370) at
../../../../../libgcc/unwind-dw2.c:202
202 return (context->flags & SIGNAL_FRAME_BIT) ? 1 : 0;
(gdb)
execute_cfa_program (insn_ptr=0x7fffffffde103 "",
insn_end=0x8000413f2246 <Address 0x8000413f2246 out of bounds>,
   context=context@entry=0x7fffffffdf370,
   fs=fs@entry=0x7fffffffdf460) at ../../../../../libgcc/unwind-dw2.c:958
958 & & fs->pc < context->ra + _Unwind_IsSignalFrame
   (context))
(gdb)
0x00000035c400d506 958 & & fs->pc < context->ra +
   _Unwind_IsSignalFrame (context))
(gdb)
0x00000035c400d509 958 & & fs->pc < context->ra +
   _Unwind_IsSignalFrame (context))
(gdb)
960 unsigned char insn = *insn_ptr++;
(gdb)
0x00000035c400d49b  960  unsigned char insn =
*insn_ptr++;
(gdb)
964  if ((insn & 0xc0) == DW_CFA_advance_loc)
(gdb)
0x00000035c400d4a1  964  if ((insn & 0xc0) ==
DW_CFA_advance_loc)
(gdb) x/20i $rip
=> 0x35c400d4a1 <execute_cfa_program+113>: and
$0xffffffffc0,%esi
0x35c400d4a4 <execute_cfa_program+116>: cmp $0x40,%sil
0x35c400d4a8 <execute_cfa_program+120>: je 0x35c400d4d0
<execute_cfa_program+160>
0x35c400d4aa <execute_cfa_program+122>: cmp $0x80,%sil
0x35c400d4ae <execute_cfa_program+126>: je 0x35c400d520
<execute_cfa_program+240>
0x35c400d4b0 <execute_cfa_program+128>: cmp $0xc0,%sil
0x35c400d4b4 <execute_cfa_program+132>: je 0x35c400d578
<execute_cfa_program+328>
0x35c400d4ba <execute_cfa_program+138>: cmp $0x2f,%al
0x35c400d4bc <execute_cfa_program+140>: ja 0x35c400d5a9
<execute_cfa_program+377>
0x35c400d4c2 <execute_cfa_program+146>: movslq
(%rdx,%rax,4),%rax
0x35c400d4c6 <execute_cfa_program+150>: add %rdx,%rax
0x35c400d4c9 <execute_cfa_program+153>: jmpq %rax
0x35c400d4cb <execute_cfa_program+155>: nopl
0x0(%rax,%rax,1)
0x35c400d4d0 <execute_cfa_program+160>: and $0x3f,%eax
0x35c400d4d3 <execute_cfa_program+163>: mov %r12,%rbx
0x35c400d4d6 <execute_cfa_program+166>: imul
0x160(%r13),%rax
0x35c400d4de <execute_cfa_program+174>: add %rcx,%rax
0x35c400d4e1 <execute_cfa_program+177>: mov
%rax,0x148(%r13)
0x35c400d4e8 <execute_cfa_program+184>: cmp %r15,%rbx
0x35c400d4eb <execute_cfa_program+187>: jae 0x35c400d50b
<execute_cfa_program+219>
(gdb) set $esi = 0xc8
(gdb) set $esi = 0x2f
...
<fix the memory to avoid killing by gcc_unreachable() or
gcc_assert()>
(gdb) 0x00000035c400d4ae 966 else if ((insn & 0xc0) == DW_CFA_offset)
(gdb) 978 else if ((insn & 0xc0) == DW_CFA_restore)
(gdb) 0x00000035c400d4b4 978 else if ((insn & 0xc0) == DW_CFA_restore)
(gdb) 985 else switch (insn)
(gdb) 0x00000035c400d4bc 985 else switch (insn)
(gdb) 0x00000035c400d4c2 985 else switch (insn)
(gdb) 0x00000035c400d4c6 985 else switch (insn)
(gdb) 0x00000035c400d4c9 985 else switch (insn)
(gdb) 960 unsigned char insn = *insn_ptr++;
(gdb) 0x00000035c400d5b1 960 unsigned char insn = *insn_ptr++;
(gdb) 957 while (insn_ptr < insn_end
(gdb) x/20i $rip => 0x35c400d4e8 <execute_cfa_program+184>: cmp %r15,%rbx
 0x35c400d4eb <execute_cfa_program+187>: jae 0x35c400d50b
<execute_cfa_program+219>
 0x35c400d4ed <execute_cfa_program+189>: mov 0xc0(%r14),%rax
 0x35c400d4f4 <execute_cfa_program+196>: mov 0x148(%r13),%rcx
 0x35c400d4fb <execute_cfa_program+203>: shr $0x3f,%rax
 0x35c400d4ff <execute_cfa_program+207>: add 0x98(%r14),%rax
 0x35c400d506 <execute_cfa_program+214>: cmp %rax,%rcx
 0x35c400d509 <execute_cfa_program+217>: jb 0x35c400d498
<execute_cfa_program+104>
 0x35c400d50b <execute_cfa_program+219>: lea -
 0x35c400d510 <execute_cfa_program+224>: pop %r12
 0x35c400d512 <execute_cfa_program+226>: pop %r13
 0x35c400d514 <execute_cfa_program+228>: pop %r14
 0x35c400d516 <execute_cfa_program+230>: pop %r15
 0x35c400d518 <execute_cfa_program+232>: pop %rbp
 0x35c400d519 <execute_cfa_program+233>: retq
 0x35c400d51a <execute_cfa_program+234>: nopw
 0x35c400d520 <execute_cfa_program+240>: mov %rax,%rdi
 0x35c400d523 <execute_cfa_program+243>: mov %r12,%rbx
0x35c400d526 <execute_cfa_program+246>: xor %esi,%esi
(gdb) i r r15 rbx
r15 0x8000413f2246 140738583011910
rbx 0x7fffffffde104 140737488216324
(gdb) set $r15 = $rbx

...<Try to jump out from the while loop because it will take
forever and requires fixing memory object for each iteration>
...
(gdb) i r r15 rbx
r15 0x7fffffffde104 140737488216324
rbx 0x7fffffffde104 140737488216324
(gdb) si
0x00000035c400d4eb 957 while (insn_ptr < insn_end
(gdb) 1224 }
(gdb)
0x00000035c400d50f 1224 }
(gdb)
0x00000035c400d510 1224 }
(gdb)
0x00000035c400d512 1224 }
(gdb)
0x00000035c400d514 1224 }
(gdb)
0x00000035c400d516 1224 }
(gdb)
0x00000035c400d518 1224 }
(gdb)
0x00000035c400d519 1224 }
(gdb)
uw_frame_state_for (context=context@entry=0x7fffffffd370,
fs=fs@entry=0x7fffffffde460) at ../../../libgcc/unwind-dw2.c:1273
1273 augment += 2 * size_of_encoded_value (fs->fde_encoding);
(gdb)
1272 augment = (const unsigned char *) fde + sizeof (*fde);
(gdb)
size_of_encoded_value (encoding=0 '\000') at
../../../libgcc/unwind-pe.h:74
74 if (encoding == DW_EH_PE_omit)
(gdb)
0x00000035c400eece 74 if (encoding == DW_EH_PE_omit)
(gdb)
77 switch (encoding & 0x07)
(gdb)
0x00000035c400eed7 77 switch (encoding & 0x07)
(gdb)
0x00000035c400eed9 77 switch (encoding & 0x07)
(gdb)
0x00000035c400eedf 77 switch (encoding & 0x07)
(gdb)
77 switch (encoding & 0x07)
0x00000035c400f51b  77  switch (encoding & 0x07)
0x00000035c400ef6  77  switch (encoding & 0x07)
uw_frame_state_for (context=context@entry=0x7fffffffffd370,
fs=fs@entry=0x7fffffffffd460) at ../../../libgcc/unwind-dw2.c:1273
  aug += 2 * size_of_encoded_value (fs->fde_encoding);
insn = NULL;
if (fs->saw_z)
0x00000035c400ef09  1275  if (fs->saw_z)
0x00000035c400ef38  1281  if (fs->lsda_encoding != DW_EH_PE_omit)
0x00000035c400ef3c  1281  if (fs->lsda_encoding !=
DW_EH_PE_omit)
next_fde (f=<optimized out>) at ../../../libgcc/unwind-dw2-fde.h:163
  return (const fde *) ((const char *) f + f->length +
sizeof (f->length));
insn = aug;
insn = aug;
insn = aug;
insn = aug;
insn = aug;
ext_fde (f=0x7ffffffffde101) at ../../../libgcc/unwind-dw2-fde.h:163
  return (const fde *) ((const char *) f + f->length +
sizeof (f->length));
insn = aug;
insn = aug;
insn = aug;
eexecute_cfa_program (insn, end, context, fs);
eexecute_cfa_program (insn, end, context, fs);
eexecute_cfa_program (insn, end, context, fs);
eexecute_cfa_program (insn, end, context, fs);

insn_end=0x7fffffffffde185 'A' <repeats 200 times>..., context=context@entry=0x7fffffffffde370, fs=fs@entry=0x7fffffffffde460)
at ../../../libgcc/unwind-dw2.c:942
942  {
(gdb)
0x00000035c400d431  942  {
(gdb)
0x00000035c400d434  942  {
(gdb)
0x00000035c400d436  942  {
(gdb)
0x00000035c400d439  942  {
(gdb)
0x00000035c400d43b  942  {
(gdb)
0x00000035c400d43d  942  {
(gdb)
0x00000035c400d440  942  {
(gdb)
0x00000035c400d442  942  {
(gdb)
0x00000035c400d443  942  {
(gdb)
0x00000035c400d446  942  {
(gdb)
957  while (insn_ptr < insn_end
(gdb)  n
946    fs->regs.prev = NULL;
(gdb)
957  while (insn_ptr < insn_end
(gdb)
958    && fs->pc < context->ra + _Unwind_IsSignalFrame
(context))
(gdb)
1169  insn_ptr = read_sleb128 (insn_ptr, &stmp);
(gdb)
985  else switch (insn)
(gdb)  x/20i $rip
=> 0x35c400d487 <execute_cfa_program+87>: lea
0x5162(%rip),%rdx  # 0x35c40125f0
 0x35c400d48e <execute_cfa_program+94>: xor %r8d,%r8d
 0x35c400d491 <execute_cfa_program+97>: mov %rax,-
0x48(%rbp)
 0x35c400d495 <execute_cfa_program+101>: nopl (%rax)
 0x35c400d498 <execute_cfa_program+104>: movzb %rbx,%eax
 0x35c400d49b <execute_cfa_program+107>: lea 0x1(%rbx),%r12
 0x35c400d49f <execute_cfa_program+111>: mov %eax,%esi
 0x35c400d4a1 <execute_cfa_program+113>: and $0xffffffff,esi
 0x35c400d4a4 <execute_cfa_program+116>: cmp $0x40,%sil
 0x35c400d4a8 <execute_cfa_program+120>: je 0x35c400d4d0
<execute_cfa_program+160>
0x35c400d4aa <execute_cfa_program+122>: cmp $0x80,%sil
0x35c400d4ae <execute_cfa_program+126>: je 0x35c400d520

<execute_cfa_program+240>
0x35c400d4b0 <execute_cfa_program+128>: cmp $0xc0,%sil
0x35c400d4b4 <execute_cfa_program+132>: je 0x35c400d578

<execute_cfa_program+328>
0x35c400d4ba <execute_cfa_program+138>: cmp $0x2f,%al
0x35c400d4bc <execute_cfa_program+140>: ja 0x35c400d5a9

<execute_cfa_program+377>
0x35c400d4c2 <execute_cfa_program+146>: movslq (%rdx,%rax,4),%rax
0x35c400d4c6 <execute_cfa_program+150>: add %rdx,%rax
0x35c400d4c9 <execute_cfa_program+153>: jmpq *%rax
0x35c400d4cb <execute_cfa_program+155>: nopl

0x0(%rax,%rax,1)
(gdb) x/x $rbx
0x7ffffffde119: 0x41
(gdb) set *$rbx = 0x80
<fix the memory to avoid killing by gcc_unreachable() or gcc_assert()>

... (gdb) si
943 struct frame_state_reg_info *unused_rs = NULL;
... (gdb) 1169 insn_ptr = read_sleb128 (insn_ptr, &stmp);
(gdb) 0x00000035c400d495 1169 insn_ptr = read_sleb128 (insn_ptr, &stmp);
... (gdb) 960 unsigned char insn = *insn_ptr++;
... (gdb) 0x00000035c400d49b 960 unsigned char insn = *insn_ptr++;
... (gdb) 964 if ((insn & 0xc0) == DW_CFA_advance_loc)
... (gdb) 0x00000035c400d4a1 964 if ((insn & 0xc0) ==
DW_CFA_advance_loc)
... (gdb) 966 else if ((insn & 0xc0) == DW_CFA_offset)
... (gdb) 0x00000035c400d4ae 966 else if ((insn & 0xc0) ==
DW_CFA_offset)
... (gdb) 968 reg = insn & 0x3f;
(gdb)
0x00000035c400d523   968   reg = insn & 0x3f;
(gdb)
read_uleb128 (val=<optimized out>, p=<optimized out>) at ../../libgcc/unwind-pe.h:137
137   result = 0;
(gdb)
execute_cfa_program (insn_ptr=0x7fffffffde11a "",
    insn_ptr@entry=0x7fffffffde119 "\200",
    insn_end=0x7fffffffde185 'A' <repeats 200 times>...,
    context=context@entry=0x7fffffffdf370, fs=fs@entry=0x7fffffffdf460)
   at ../../libgcc/unwind-dw2.c:968
968   reg = insn & 0x3f;
(gdb)
read_uleb128 (val=<synthetic pointer>, p=0x
7fffffffde11a "") at ../../libgcc/unwind-pe.h:133
133   unsigned int shift = 0;
(gdb)
0x00000035c400d52d   133   unsigned int shift = 0;
(gdb)
140   byte = *p++;
(gdb)
0x00000035c400d534   140   byte = *p++;
(gdb)
141   result |= ((_uleb128_t)byte & 0x7f) << shift;
(gdb)
0x00000035c400d53c   141   result |= ((_uleb128_t)byte &
0x7f) << shift;
(gdb)
0x00000035c400d53f   141   result |= ((_uleb128_t)byte &
0x7f) << shift;
(gdb)
142   shift += 7;
(gdb)
141   result |= ((_uleb128_t)byte & 0x7f) << shift;
(gdb)
144   while (byte & 0x80);
(gdb)
0x00000035c400d54b   144   while (byte & 0x80);
(gdb)
execute_cfa_program (insn_ptr=0x7fffffffde11b "",
    insn_ptr@entry=0x7fffffffde119 "\200",
    insn_end=0x7fffffffde185 'A' <repeats 200 times>...,
    context=context@entry=0x7fffffffdf370, fs=fs@entry=0x7fffffffdf460)
   at ../../libgcc/unwind-dw2.c:970
970   offset = (_Unwind_Sword) utmp * fs->data_align;
(gdb)
972   if (UNWIND_COLUMN_IN_RANGE (reg))
(gdb)
0x00000035c400d559   972   if (UNWIND_COLUMN_IN_RANGE (reg))
(gdb)
if (UNWIND_COLUMN_IN_RANGE (reg))
(gdb)
if (UNWIND_COLUMN_IN_RANGE (reg))
(gdb)
fs->regs.reg[reg].how = REG_SAVED_OFFSET;
(gdb)
fs->regs.reg[reg].loc.offset = offset;
(gdb)
0x00000035c400d56c 975  fs->
>regs.reg[reg].loc.offset = offset;
(gdb)
957  while (insn_ptr < insn_end
(gdb)
0x00000035c400d4eb 957  while (insn_ptr < insn_end
(gdb) x/20i $rip=10
0x35c400d4e1 <execute_cfa_program+177>: mov
%rax,0x148(%r13)
0x35c400d4e8 <execute_cfa_program+184>: cmp %r15,%rbx
=> 0x35c400d4eb <execute_cfa_program+187>: jae 0x35c400d50b
<execute_cfa_program+219>
0x35c400d4ed <execute_cfa_program+189>: mov
0xc0(%r14),%rax
0x35c400d4f <execute_cfa_program+196>: mov
0x148(%r13),%rcx
0x35c400d4fb <execute_cfa_program+203>: shr $0x3f,%rax
0x35c400d4ff <execute_cfa_program+207>: add
0x98(%r14),%rax
0x35c400d506 <execute_cfa_program+214>: cmp %rax,%rcx
0x35c400d509 <execute_cfa_program+217>: jb 0x35c400d498
<execute_cfa_program+104>
0x35c400d50b <execute_cfa_program+219>: lea -
0x28(%rbp),%rsp
0x35c400d50f <execute_cfa_program+223>: pop %rbx
0x35c400d510 <execute_cfa_program+224>: pop %r12
0x35c400d512 <execute_cfa_program+226>: pop %r13
0x35c400d514 <execute_cfa_program+228>: pop %r14
0x35c400d516 <execute_cfa_program+230>: pop %r15
0x35c400d518 <execute_cfa_program+232>: pop %rbp
0x35c400d519 <execute_cfa_program+233>: retq
0x35c400d51a <execute_cfa_program+234>: nopw
0x0(%rax,%rax,1)
0x35c400d520 <execute_cfa_program+240>: mov %rax,%rdi
0x35c400d523 <execute_cfa_program+243>: mov %r12,%rbx
(gdb) i r r15 rbx
r15  0x7fffffffde185  140737488216453
rbx  0x7fffffffde11b  140737488216347
(gdb) set $r15 = $rbx
(gdb) set $rip = 0x35c400d4e8

...
<fix while loop again... <- its 2nd entry to this function (for upper FDE)>

... 
(gdb) si
0x00000035c400d4eb 957 while (insn_ptr < insn_end
(gdb) 
1224 )
(gdb)
0x00000035c400d50f 1224 }
(gdb)
0x00000035c400d510 1224 }
(gdb)
0x00000035c400d512 1224 }
(gdb)
0x00000035c400d514 1224 }
(gdb)
0x00000035c400d516 1224 }
(gdb)
0x00000035c400d518 1224 }
(gdb)
0x00000035c400d519 1224 }
(gdb)
uw_frame_state_for (context=context@entry=0x7fffffffd370,
fs=fs@entry=0x7fffffffd460) at ../../../libgcc/unwind-dw2.c:1296
1296 }
(gdb) list
1291     insn = aug;
1292     end = (const unsigned char *) next_fde (fde);
1293     execute_cfa_program (insn, end, context, fs);
1294     return _URC_NO_REASON;
1296 }
(gdb) ^L
1298 typedef struct frame_state
1299 {
1300     void *cfa;
(gdb)
1301     void *eh_ptr;
1302     long cfa_offset;
1303     long args_size;
1304     long reg_or_offset[PRE_GCC3_DWARF_FRAME_REGISTERS+1];
1305     unsigned short cfa_reg;
1306     unsigned short retaddr_column;
1307     char saved[PRE_GCC3_DWARF_FRAME_REGISTERS+1];
1308 } frame_state;
1309
1310 struct frame_state *__frame_state_for (void *, struct
frame_state *);
(gdb) si
1295     return _URC_NO_REASON;
(gdb)
1296 }

82
(gdb)
0x00000035c400f486 1296 }
(gdb)
0x00000035c400f487 1296 }
(gdb)
0x00000035c400f489 1296 }
(gdb)
0x00000035c400f48b 1296 }
(gdb)
0x00000035c400f48d 1296 }
(gdb)
0x00000035c400f48f 1296 }
(gdb)
_UUnwind_Backtrace (trace=0x35c1909bc0 <backtrace_helper>,
trace_argument=0x7fffffffd620) at ../../../libgcc/unwind.inc:291
291 if (code != _URC_NO_REASON && code !=
_URC_END_OF_STACK)
(gdb) code = uw_frame_state_for (&context, &fs);
(gdb) n
291 if (code != _URC_NO_REASON && code !=
_URC_END_OF_STACK)
(gdb) print code
$157 = _URC_NO_REASON
(gdb) n
295 if ((*trace) (&context, trace_argument) !=
_URC_NO_REASON)
(gdb) code = uw_frame_state_for (&context, &fs);
(gdb) si
0x00000035c400ff06 303 uw_update_context (&context, &fs);
(gdb) n
Program received signal SIGABRT, Aborted.
0x00000035c1835a19 in __GI_raise (sig=sig@entry=6) at
../nptl/sysdeps/unix/sysv/linux/raise.c:56
56 return INLINE_SYSCALL (tgkill, 3, pid, selftid, sig);
(gdb) bt
#0 0x00000035c1835a19 in __GI_raise (sig=sig@entry=6) at
../nptl/sysdeps/unix/sysv/linux/raise.c:56
#1 0x00000035c1837128 in __GI_abort () at abort.c:90
#2 0x00000035c400e8d5 in uw_update_context_1
(context=context@entry=0x7fffffffd370,
fs=fs@entry=0x7fffffffd460)
at ../nptl/sysdeps/unix/sysv/linux/raise.c:56

#3  0x00000035c400eb61 in uw_update_context
  (context=context@entry=0x7fffffffd370, 
   fs=fs@entry=0x7fffffffd460)
  at ../../../libgcc/unwind-dw2.c:1506
#4  0x00000035c400ff0e in _Unwind_Backtrace (trace=0x35c1909bc0
  <backtrace_helper>, trace_argument=0x7fffffffd620)
  at ../../../libgcc/unwind.inc:303
#5  0x00000035c1909d36 in _GI__backtrace
  (array=array@entry=0x7fffffffd800, size=size@entry=64)
  at ../sysdeps/x86_64/backtrace.c:109
#6  0x00000035c1875d64 in __libc_message
  (do_abort=do_abort@entry=2,
   fmt=fmt@entry=0x35c197d302 "*** %s ***: %s terminated\n") at 
  ../sysdeps/unix/sysv/linux/libc_fatal.c:176
#7  0x00000035c190d6b7 in __GI___fortify_fail
  (msg=msg@entry=0x35c197d2ea "stack smashing detected") at
  fortify_fail.c:31
#8  0x00000035c190d680 in __stack_chk_fail () at
  stack_chk_fail.c:28
#9  0x00000035c1837128 in __GI_abort () at abort.c:90
  (gdb) up
#1  0x00000035c1837128 in __GI_abort () at abort.c:90
  raise (SIGABRT);
  (gdb)
#2  0x00000035c400e8d5 in uw_update_context_1
  (context=context@entry=0x7fffffffd370, 
   fs=fs@entry=0x7fffffffd460)
  at ../../../libgcc/unwind-dw2.c:1430
1430__gcc_unreachable (); <- one of the object wasn’t
  patched ;)
(gdb) print fs->regs.cfa_how
$159 = CFA_UNSET
(gdb) print *fs
$160 = {regs = {reg = {{loc = {reg = 0, offset = 0, exp = 0x0}, 
   how = REG_SAVED_OFFSET}, {loc = {reg = 0, offset = 0, 
   exp = 0x0}, how = REG_UNSAVED} <repeats 17 times>}, 
   prev = 0x0, cfa_offset = 0, cfa_reg = 0, cfa_exp = 0x0, 
   cfa_how = CFA_UNSET}, pc = 0x400630 <main>, personality = 
   0x0, data_align = 0, code_align = 0, retaddr_column = 0, 
   fde_encoding = 0 '\000', lsda_encoding = 255 '\377', saw_z = 0 
   '\000', signal_frame = 0 '\000', eh_ptr = 0x0}
Somehow security related:

Scenario 1 – force SSP to recopy existing process’s memory region to the newly allocated one. It is very difficult because to hit vulnerable code, you need to fix all possible crashes which I’ve described before (and reproduced). This mean fixing environment block, pointers to argument etc. I’ve switched off ASLR + did some memory patching just to monitor control flow and to see possible memory exhaustion. In my scenario I’ve allocated 2GB of memory where I've pointed as pointer to the program's name.

[pi3@localhost ~]$ gdb -q -p 13633
Attaching to process 13633
Reading symbols from /home/pi3/test...done.
Reading symbols from /lib64/libc.so.6...Reading symbols from
/usr/lib/debug/lib64/libc-2.17.so.debug...done.
done.
Loaded symbols for /lib64/libc.so.6
Reading symbols from /lib64/ld-linux-x86-64.so.2...Reading
symbols from /usr/lib/debug/lib64/ld-2.17.so.debug...done.
done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
0x00000035c17e78a0 in __read_nocancel () at
../sysdeps/unix/syscall-template.S:81
81  T_PSEUDO (SYSCALL_SYMBOL, SYSCALL_NAME, SYSCALL_NARGS)
(gdb) b libc_fatal.c:66
Breakpoint 1 at 0x35c1875a37: file
../sysdeps/unix/sysv/linux/libc_fatal.c, line 66.
(gdb) c
Continuing.

Breakpoint 1, __libc_message (do_abort=do_abort@entry=2,
fmt=fmt@entry=0x35c197d302 "*** %s ***: %s terminated\n")
at ../sysdeps/unix/sysv/linux/libc_fatal.c:66
66    const char *on_2 = __libc_secure_getenv
("LIBC_FATAL_STDERR_");
(gdb) ni
57    va_start (ap, fmt);
(gdb) 0x00000035c1875a45  57    va_start (ap, fmt);
(gdb) 0x00000035c1875a4c  57    va_start (ap, fmt);
(gdb) 0x00000035c1875a56  57    va_start (ap, fmt);
(gdb) 52  {
(gdb) 57    va_start (ap, fmt);
(gdb) 58    va_copy (ap_copy, ap);
(gdb) 0x00000035c1875a75  58    va_copy (ap_copy, ap);
0x00000035c1875a7c 58 va_copy (ap_copy, ap);
(gdb)
0x00000035c1875a80 58 va_copy (ap_copy, ap);
(gdb)
0x00000035c1875a87 58 va_copy (ap_copy, ap);
(gdb)
0x00000035c1875a8e 58 va_copy (ap_copy, ap);
(gdb)
66 const char *on_2 = __libc_secure_getenv("LIBC_FATAL_STDERR_");
(gdb)
67 if (on_2 == NULL || *on_2 == '\0')
(gdb) si
0x00000035c1875a9d 67 if (on_2 == NULL || *on_2 == '\0')
(gdb) x/20i 0x00000035c1875a9d
=> 0x35c1875a9d <__libc_message+205>: je 0x35c1875aa8
<__libc_message+216>
0x35c1875a9f <__libc_message+207>: cmpb $0x0, (%rax)
0x35c1875aa2 <__libc_message+210>: jne 0x35c1875c68
<__libc_message+664>
0x35c1875aa8 <__libc_message+216>: lea 0x1061c6(%rip), %rdi # 0x35c197bc75
0x35c1875aaf <__libc_message+223>: xor %eax, %eax
0x35c1875ab1 <__libc_message+225>: mov $0x902, %esi
0x35c1875ab6 <__libc_message+230>: callq 0x35c18e7459
<__open_nocancel>
0x35c1875abb <__libc_message+235>: cmp $0xffffffff, %eax
0x35c1875abe <__libc_message+238>: mov %eax, -0x718(%rbp)
0x35c1875ac4 <__libc_message+244>: je 0x35c1875c68
<__libc_message+664>
0x35c1875aca <__libc_message+250>: mov -0x720(%rbp), %rax
0x35c1875ad1 <__libc_message+257>: xor %r14d, %r14d
0x35c1875ad4 <__libc_message+260>: xor %r13d, %r13d
0x35c1875ad7 <__libc_message+263>: movzbl (%rax), %r12d
0x35c1875adb <__libc_message+267>: mov %rax, %rbx
0x35c1875ade <__libc_message+270>: test %r12b, %r12b
0x35c1875ae1 <__libc_message+273>: je 0x35c1875c8e
<__libc_message+702>
0x35c1875ae7 <__libc_message+279>: nopw 0x0(%rax, %rax, 1)
0x35c1875af0 <__libc_message+288>: mov %r12d, %edx
0x35c1875af3 <__libc_message+291>: mov %rbx, %rax
(gdb) i r rax
rax 0x7fff8b4a4fe0 140735530291168
(gdb) x/x $rax
0x7fff8b4a4fe0: 0x706d742f
(gdb) si
0x00000035c1875a9f 67 if (on_2 == NULL || *on_2 == '\0')
(gdb) si
0x00000035c1875aa2 67 if (on_2 == NULL || *on_2 == '\0')
(gdb) x/10i 0x35c1875c68
0x35c1875c68 <__libc_message+664>: movl $0x2, -0x718(%rbp)
0x35c1875c72 <__libc_message+674>:   jmpq 0x35c1875aca
__libc_message+250>
0x35c1875c77 <__libc_message+679>:   mov -0x708 (%rbp),%rax
0x35c1875c7e <__libc_message+686>:   lea 0x8 (%rax),%rdx
0x35c1875c82 <__libc_message+690>:   mov %rdx,-0x708 (%rbp)
0x35c1875c89 <__libc_message+697>:   jmpq 0x35c1875b8f
__libc_message+447>
0x35c1875c8e <__libc_message+702>:   mov -0x720 (%rbp),%rsi
0x35c1875c95 <__libc_message+709>:   lea -0x6f8 (%rbp),%rdx
0x35c1875c9c <__libc_message+716>:   mov $0x3,%edi
0x35c1875ca1 <__libc_message+721>:   callq 0x35c18f0430
__vsyslog>
(gdb) i r rip
rip 0x35c1875c68 0x35c1875c68 <__libc_message+664>
(gdb) bt
#0 __libc_message (do_abort=do_abort@entry=2,
fmt=fmt@entry=0x35c197d302 "*** %s ***: %s terminated\n")
at ../sysdeps/unix/sysv/linux/libc_fatal.c:71
#1 0x00000035c190d6b7 in __GI___fortify_fail
(msg=msg@entry=0x35c197d2ea "stack smashing detected") at
fortify_fail.c:31
#2 0x00000035c190d680 in __stack_chk_fail () at
stack_chk_fail.c:28
#3 0x00000035c190d680 in __stack_chk_fail () at
test.c:15
...
...<bla bla bla>
...
(gdb) c
Continuing.
Breakpoint 4, __libc_message (do_abort=do_abort@entry=2,
fmt=fmt@entry=0x35c197d302 "*** %s ***: %s terminated\n")
at ../sysdeps/unix/sysv/linux/libc_fatal.c:95
95 len = strlen (str);
(gdb) print str
$5 = 0x35c197d2ea "stack smashing detected"
(gdb) c
Continuing.
Breakpoint 3, __libc_message (do_abort=do_abort@entry=2,
fmt=fmt@entry=0x35c197d302 "*** %s ***: %s terminated\n")
at ../sysdeps/unix/sysv/linux/libc_fatal.c:106
106 newp->str = str;
...
...<bla bla bla>
...
(gdb) n
0x00000035c1875b9a 95 len = strlen (str);
(gdb)
96 cp += 2;
(gdb) print len
$11 = -2147483647

For some reason my gdb has small bug here:

(gdb) print len
$11 = -2147483647

Program was correctly compiled with debug symbols (flags: "-ggdb" and "-g") but gdb could find correct definition of "len" variable and dumped it as integer (which is the default type in case when gdb don’t know the type).

Greetings:

Mateusz 'j00ru' Jurczyk, Gynvael Coldwind – for reviewing and great confrontation (discussion) of ideas.
Rafał 'nergal' Wojtczuk – for great discussion and helping with DWARF.
Brian Pak – for great research and at least for reading this paper ;)
sergio and #trololol – for motivation and at least for reading this paper ;)

References:


Best regards,
Adam 'pi3' Zabrocki

http://pi3.com.pl