

[< 返回博客 \(/c/blog/\)](#)

安全RAM：通过校验和保护内存部分

2022 年 1 月 27 日 | 作者: [卢卡斯·兰斯基](https://www.suse.com/c/author/lanskysuse-com/)
(<https://www.suse.com/c/author/lanskysuse-com/>)

分享

这篇特别的博客文章不会讨论 Linux 内存管理及其安全性或如何编写安全关键软件，而是会涉及所有安全关键软件必须正确解决的主题。本主题是免于干扰 (FFI)。在我们深入讨论之前，让我们定义一些术语以及它们在现实中的含义：

ISO26262	汽车功能安全标准	这为您提供了一些指导，您应该如何制作安全关键软件。至少在汽车领域是这样。如果您主要关心软件，那么该标准的第 6 部分是最重要的。
质量管理	质量管理软件	不是安全关键软件。它可以（并且将会）发病并开始写入内存或停止 CPU。
ASIL AD	汽车软件完整性级别	汽车安全关键软件。这个软件也可能会发病。但这种情况发生的概率是有限的。限制的程度取决于完整性级别，其中 ASIL D 是最高完整性。

ISO26262 第 6 部分将不受干扰定义为软件组件之间不存在级联故障。换句话说，如何确保安全关键组件按设计运行，而不给其他组件（不太关键的组件）影响执行的机会。一般来说，干扰有 3 种类型：空间（内存）、时间（调度）和通信（输入和输出数据损坏）。正如标题所示，我们将仅详细讨论一种特定的独立于硬件的内存保护机制。

但总的来说，当涉及到干扰的空间自由度时，我们还可以想到更多的机制：

1. 使用校验和保护内存区域
2. 通过其补码的双重存储来保护每个变量
3. 在操作系统、基础软件或硬件层实现的内存保护机制



如前所述，这篇文章主要是关于 (1) – 使用校验和保护进程的给定内存部分。这将在 Linux 操作系统的示例中进行说明，但本质上（正如您稍后将看到的）它更适合更多的裸机应用程序。在所有环境中，应假设任何其他 ASIL（或 QM）低于组件所需完整性级别的进程都可能（并且将会）错误执行，并且将会（如果有机会）更改您的内存。通常我们无法阻止这种情况的发生，但我们至少可以确保及时发现并采取纠正措施。

但事不宜迟，让我们看一下这个例子。[此处](https://github.com/llansky3/safram)

[\(https://github.com/llansky3/safram\)](https://github.com/llansky3/safram)显示的所有源代码（以及更多）都可以在 GitHub 上找到。

此示例的特定目标是使用校验和保护进程的专用内存部分。这将使我们能够检测任何入侵并重置进程，以防发生内存损坏。然而，这里有一个重要的假设：在代码的安全关键部分不会发生抢占，并且没有其他代码可以并行执行（单核 CPU）。该部分由以下函数表示：

th_safram_crc.c – 安全关键部分

```
void periodic_task(char **argv, uint8_t count)
{
    /* Start of safety critical section */
    uint8_t crc;
    if (safram_crc_check()) {
        restart(argv, count);
    }
    testarray[0]++;
    crc = safram_crc_protect();
    /* End of safety critical section */

    printf("testarray[0]=%u, checksum=%u\n", testarray[0], crc);
    sleep(1);
}
```

另请注意，堆栈（或实际上除“.data_safram”之外的任何其他内存部分）不受保护。因此，如果这是在具有简单抢占式调度的裸机上运行，则不会实现干扰的空间自由度，因为任何其他进程都可能在执行安全关键部分期间更改内存（例如，想象其他进程在校验和完成后立即更改内存）检查）并且没有任何机制可以检测到这一点（实际上我们甚至会在运行结束时存储根据错误数据计算的新校验和并继续）。

但让我们稍微回顾一下——第一步是在链接描述文件中定义内存部分。在我们的示例中，我们将此部分称为“.data_safram”：

data_safram.ld – 链接器脚本 – 定义自定义内存部分



```
SECTIONS
{
    .data_safram : {
        _DATA_SAFRAM_START = .;
        *(.data_safram);
        . = . + 1;
        _DATA_SAFRAM_END = .;
    }
}
INSERT AFTER .data
```

请注意，“. = . + 1”在本节末尾添加了一个字节。这是我们存储校验和的地方。然后可以使用“-T”选项将此链接器脚本传递给链接器（如果使用 gcc）：

Makefile – 链接器选项

```
gcc --std=c99 -Wall safram_crc.o crc8.o -o th_safram_crc th_safram_crc.c -Wl,-v -T data_safram.ld
```

请注意，这种方式使用默认链接描述文件，唯一的变化是新添加的内存部分。然后我们可以开始将变量添加到此部分：

th_safram_crc.c – 安全关键变量

```
__attribute__((section(".data_safram"))) uint8_t testarray[] = {0x33, 0x22, 0x55, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF};
```

要检查我们的内存部分是否正确定义，我们可以运行：

readelf – 内存部分



```
> readelf -a -W th_safram_crc
...
Section Headers:
 [Nr] Name                Type              Address            Off    Size   ES Flg Lk Inf Al
 [ 0]                      NULL              0000000000000000  000000 000000 00      0  0  0
 [ 1] .interp                PROGBITS          0000000000400238  000238 00001c 00     A  0  0  1
...
 [14] .text                  PROGBITS          00000000004005a0  0005a0 000402 00    AX  0  0 16
 [15] .fini                  PROGBITS          00000000004009a4  0009a4 000009 00    AX  0  0  4
 [16] .rodata                PROGBITS          00000000004009c0  0009c0 0001f4 00     A  0  0 32
 [17] .eh_frame_hdr          PROGBITS          0000000000400bb4  000bb4 00005c 00     A  0  0  4
 [18] .eh_frame              PROGBITS          0000000000400c10  000c10 000190 00     A  0  0  8
 [19] .init_array             INIT_ARRAY        0000000000600e00  000e00 000008 08    WA  0  0  8
 [20] .fini_array            FINI_ARRAY        0000000000600e08  000e08 000008 08    WA  0  0  8
 [21] .dynamic               DYNAMIC           0000000000600e10  000e10 0001e0 10    WA  7  0  8
 [22] .got                   PROGBITS          0000000000600ff0  000ff0 000010 08    WA  0  0  8
 [23] .got.plt               PROGBITS          0000000000601000  001000 000048 08    WA  0  0  8
 [24] .data                  PROGBITS          0000000000601048  001048 000010 00    WA  0  0  8
 [25] .data_safram           PROGBITS          0000000000601058  001058 00000a 00    WA  0  0  8
 [26] .bss                   NOBITS            0000000000601062  001062 000006 00    WA  0  0  1
...
Key to Flags:
 W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
 L (link order), O (extra OS processing required), G (group), T (TLS),
 C (compressed), x (unknown), o (OS specific), E (exclude),
 D (mbind), l (large), p (processor specific)

...
```

链接器将提供我们可以用来了解“.data_safram”内存部分开始和结束位置的地址。在代码中，这些是以下“外部”变量：

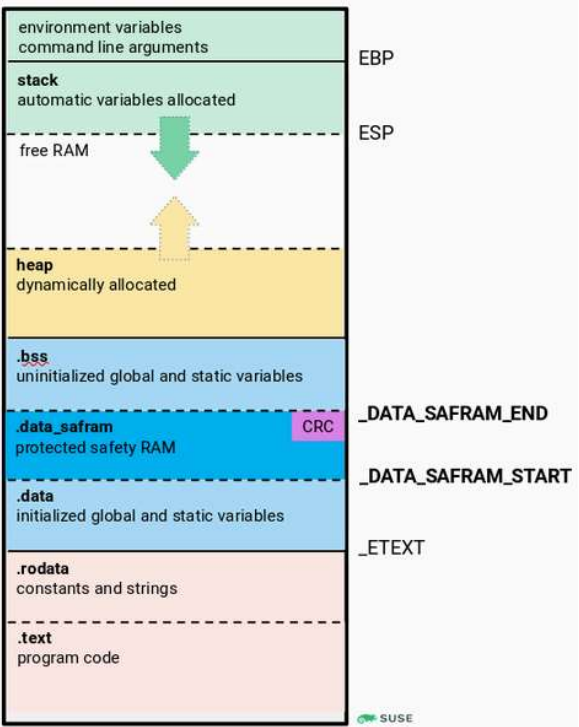
safram_crc.h – 安全 RAM 的起始和结束地址

```
extern uint32_t _DATA_SAFRAM_START;
extern uint32_t _DATA_SAFRAM_END;

#define PTR_UINT8_SAFRAM_START (uint8_t*)&_DATA_SAFRAM_START
#define PTR_UINT8_SAFRAM_END   (uint8_t*)&_DATA_SAFRAM_END
```

为了说明这一点，我们的应用程序的内存布局如下图所示。这张图就是受此启发 (https://www.ele.uva.es/~jesus/hardware_empotrado/Compiler.pdf)。





一切就绪后，我们终于可以实现存储（保护）和检查校验和的功能了。使用的校验和是 J1850 CRC-8，如**CRC 例程的**
(https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_CRCLibrary.pdf)AUTOSAR
(https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_CRCLibrary.pdf)规范中所述。
CRC 方法在这里很好，因为它旨在防止随机错误，而不是针对我们软件的恶意攻击。然而，对于大型数据集，人们可能希望用 CRC-16 或 CRC-32 替换 CRC-8。

safram_crc.c – 保护和检查安全性的函数



```
uint8_t safram_crc_protect(void)
{
    uint8_t *safram_start = PTR_UINT8_SAFRAM_START;
    uint8_t *safram_end = PTR_UINT8_SAFRAM_END;
    uint8_t crc;

    /* calculate CRC - excluding checksum itself */
    crc = crc8(safram_start, (uint32_t) (safram_end - safram_start - 1), 0xFF);
    crc = crc ^ 0xFF;

    /* last byte in .data_safram memory section is the checksum */
    *safram_end = crc;
    return crc;
}

uint8_t safram_crc_check(void)
{
    uint8_t *safram_start = PTR_UINT8_SAFRAM_START;
    uint8_t *safram_end = PTR_UINT8_SAFRAM_END;
    uint8_t crc;

    crc = crc8(safram_start, (uint32_t) (safram_end - safram_start - 1), 0xFF);
    crc = crc ^ 0xFF;

    if (*safram_end != crc)
    {
        /* Memory is corrupted */
        return 1;
    }
    else
    {
        /* All good, checksum matches */
        return 0;
    }
}
```

然后可以在我们的安全关键部分的开头和结尾使用它们 - 请参见上文。接下来的问题是, 如果检测到内存损坏该怎么办。在简单的硬件中, 这将是一个重置, 在我们的示例中, 我们将重新启动该过程, 希望一切都能恢复正常:

th_safram_crc.c - Linux 中重新启动进程

```
void restart(char **argv, uint8_t count)
{
    char buffer[32] = {};
    char *nargv[] = {argv[0], buffer, 0};

    snprintf(buffer, sizeof(buffer), "%d", count + 1);
    execv("/proc/self/exe", nargv);
    printf("This should never be reached!");
}
```



这段代码的灵感来自于这个答案
(<https://unix.stackexchange.com/a/208003>)。

那么让我们测试一下这一切。为此，我们还需要一种破坏数据的方法。幸运的是，在 Linux 中，这可以通过使用正确的权限更改“/proc/\$pid/mem”来相当容易地完成：

th_safram_corruption.c – 访问 Linux 中的其他进程内存

```
uint8_t *data = malloc(length);

lseek(fd_proc_mem, start_address, SEEK_SET);
read(fd_proc_mem, data, length);

printf("Data at 0x%x in process %d is:", start_address, pid);
for (uint16_t i = 0; i < length; i++ ) {
    if (!(i % 10)) {
        printf("\n");
    }
    printf("  %X\n", data[i]);
}

printf("Adding one to the first byte!\n");
data[0]++;

lseek(fd_proc_mem, start_address, SEEK_SET);
if (write (fd_proc_mem, data, length) == -1) {
    printf("Error while writing\n");
    exit(1);
}

free(data);
```

这段代码很大程度上受此启发
(<https://renenyffenegger.ch/notes/Linux/memory/read-write-another-processes-memory>)。

在这里您可以看到结果- “...”行只是试图显示时间对齐。

流程 1 – ASIL	流程 2 – 质量管理
th_safram_crc	th_safram_corruption




```
> ./th_safram_crc
This is an example of protecting RAM area with
checksum! Run #0!
pid=22808, safram start_address=0x601058, leng
th=9, checksum=203
ptrace restrictions are effectively disabled
testarray[0]=34, checksum=237
testarray[0]=35, checksum=136
testarray[0]=36, checksum=39
testarray[0]=37, checksum=66
testarray[0]=38, checksum=107
testarray[0]=39, checksum=14
testarray[0]=3A, checksum=161
This is an example of protecting RAM area with
checksum! Run #1!
pid=22808, safram start_address=0x601058, leng
th=9, checksum=203
testarray[0]=34, checksum=237
testarray[0]=35, checksum=136
testarray[0]=36, checksum=39
...
...
...
testarray[0]=37, checksum=66
testarray[0]=38, checksum=107
This is an example of protecting RAM area with
checksum! Run #2!
pid=22808, safram start_address=0x601058, leng
th=9, checksum=203
testarray[0]=52, checksum=237
testarray[0]=53, checksum=136
testarray[0]=54, checksum=39
testarray[0]=55, checksum=66
testarray[0]=56, checksum=107
testarray[0]=57, checksum=14
```

```
...
...
...
...
...
...
...
...
...
> ./th_safram_corruption 22808 0x601058 4
Opening /proc/22808/mem, address is 0x601058
Data at 0x601058 in process 22808 is:
  3A
  22
  55
  AA
Adding one to the first byte!
...
...
...
...
> ./th_safram_corruption 22808 0x601058 4
Opening /proc/22808/mem, address is 0x601058
Data at 0x601058 in process 22808 is:
  38
  22
  55
  AA
Adding one to the first byte!
```

您可以看到，每当我们损坏进程 2 的内存时，进程 1 就会重新启动。如果不这样做，那么您的 Linux 发行版的配置方式可能会阻止您写入其他进程内存。但这是未来帖子的主题。然而，这个示例在[openSUSE Leap 15](https://www.opensuse.org/#Leap) (<https://www.opensuse.org/#Leap>)上运行良好。

所以它是有效的，但需要明确的是，这里的过程 1 根本不是 ASIL。首先，我没有遵循 ISO26262 要求的任何编码实践，但即使我这样做了，这里无论如何也不满足有关抢占的假设。因此，请以此为例，错误（不合适）的技术概念不会导致软件安全，即使该软件完全符合 ISO26262 标准。最重要的是，我们在非安全的 Linux 操作系统上运行，因此我们的进程重置机制不可靠，并且没有任何东西可以确保我们的检查得到执行。所以抱歉，这里根本没有 ASIL。

总而言之，如果我们需要在函数运行（例如状态）执行之间保护全局静态变量，我们可以确保安全关键部分不会被安全关键性较低的软件组件抢占，那么这种方法会很方便。这种方法的优点是它主要是关于链接器的，其他代码需要修改很少，而且也

不需要太多额外的内存或计算能力。缺点是我们常常想要或必须允许抢占和并行。在这种情况下我们必须使用其他方法。如果您想了解补集的双存储或者 Linux 的内存管理实现了哪些保护机制，请继续关注。

如前所述，所有代码都在这里：<https://github.com/llansky3/safram> (<https://github.com/llansky3/safram>) 在“crc”文件夹中。

更新 (2022/04/12)：现在有后续博客文章：[安全 RAM：用补码保护变量](https://www.suse.com/c/safety-ram-protecting-variables-with-ones-complement) (<https://www.suse.com/c/safety-ram-protecting-variables-with-ones-complement>)。特别是“内存损坏问题” (<https://www.suse.com/c/safety-ram-protecting-variables-with-ones-complement/#MemoryCorruptionProblems>) 部分是这篇博文直接后续内容。

相关文章

2023 年 10 月 31 日

机密云：机密计算简介 (<https://www.suse.com/c/introduction-to-confidential-cloud-computing/>)

劳尔·马赫克斯 (<https://www.suse.com/c/author/rmahiquessuse-com/>)

2022 年 7 月 5 日

在时尚朋友的帮助下写作 (<https://www.suse.com/c/write-with-a-little-help-from-a-stylish-friend/>)

梅克·查博夫斯基 (<https://www.suse.com/c/author/chabowski/>)

2024 年 3 月 19 日

认识 Rancher Prime 3.0：您的平台工程团队的新好朋友
(<https://www.suse.com/c/meet->

史黛西·米勒
(https://www.suse.com/c/author/stacey_miller/)



2022 年 4 月 27 日

加入我们的 HPE Discover 2022 (<https://www.suse.com/c/join-us-at-hpe-discover-2022/>)

李亚伦 (<https://www.suse.com/c/author/aaron-leesuse-com/>)

发表评论

您的电子邮件地址不会被公开。 必需的地方已做标记*

评论*

姓名*

电子邮件*

网站

☐ 在此浏览器中保存我的姓名、电子邮件和网站，以便下次发表评论时使用。

发表评论

暂时没有评论



4,718 次观看

Lukas Lansky
(<https://www.suse.com/c/author/llanskys-com/>)汽车 Linux 流程经理

