如何安全清理隐私数据？

我们经常需要在程序中存储隐私数据，例如密码、密钥和它们的派生物，我们通常需要在使用之后清理它们在内存中的痕迹，从而使潜在的入侵者无法获得这些数据。在这篇文章中我们讨论了为什么不能用函数 memset()清理隐私数据。

memset()

您可能已经在其他文章中了解到在程序中使用 memset() 清理内存是存在漏洞的。 然而，这篇文章并没有完全考虑 memset() 使用不当导致出现的所有情况。您遇到的问题不仅与清理堆栈分配的缓冲区相关，同样也与清理动态分配的缓冲区相关。

The stack

首先，让我们讨论上述提到的文章中的一个例子，该例是使用栈分配变量进行处理。

下面是一段处理密码的代码：

```
#include <string>
#include <functional>
```

```
#include <iostream>

//Private data
struct PrivateData
{
   size_t m_hash;
   char m_pswd[100];
};

//Function performs some operations on password
void doSmth(PrivateData& data)
{
   std::string s(data.m_pswd);
   std::hash<std::string> hash_fn;

   data.m_hash = hash_fn(s);
}

//Function for password entering and processing
int funcPswd()
{
   PrivateData data;
```

```
    std::cin >> data.m_pswd;


    doSmth(data);
    memset(&data, 0, sizeof(PrivateData));
    return 1;
}


int main()
{
    funcPswd();
    return 0;
}
```

这个例子是相当传统且完全合成的。

如果我们构建了一个调试版本的代码，并在调试器中运行它（我使用的是Visual Studio2015），我们会看到改代码运行良好：密码和其计算的哈希值都在使用后被删除。

我们来看一下代码在 Visual Studio 调试器中的汇编版本：

```
....
        doSmth(data);
000000013F3072BF    lea             rcx,[data]
000000013F3072C3    call            doSmth (013F30153Ch)
    memset(&data, 0, sizeof(PrivateData));
```

```
000000013F3072C8    mov             r8d,70h
000000013F3072CE    xor             edx,edx
000000013F3072D0    lea             rcx,[data]
000000013F3072D4    call            memset (013F301352h)
   return 1;
000000013F3072D9    mov             eax,1
....
```

我们看出调用 memset() 函数，可以在使用私人数据后清理。

我们可以停在这里，但我们将继续尝试建立一个优化的版本。这时，我们在调试器中可以看到：

```
....
000000013F7A1035    call
            std::operator>><char,std::char_traits<char> >
(013F7A18B0h)
000000013F7A103A    lea             rcx,[rsp+20h]
000000013F7A103F    call            doSmth (013F7A1170h)
      return 0;
000000013F7A1044    xor             eax,eax
....
```

所有调用 memset() 的相关指令已被删除。编译器认为没有必要调用函数清理数据，因为它们已经不再使用。这并不是一个错误，这是编译器的合法选择。从语言角度来看，不需要调用 memset() 是由于程序中缓冲区不再进一步使用，

这样移除该调用并不会影响其行为。所以我们的隐私数据仍未清理，这是非常糟糕的。

The heap

现在让我们理解更深入些。看一下当我们在动态内存中使用 malloc 函数或 new 运算符分配数据时，数据会发生什么变化。

让我们修改之前的 malloc 代码：

```
#include <string>
#include <functional>
#include <iostream>

struct PrivateData
{
   size_t m_hash;
   char m_pswd[100];
};

void doSmth(PrivateData& data)
{
   std::string s(data.m_pswd);
   std::hash<std::string> hash_fn;

   data.m_hash = hash_fn(s);
```

```
}

int funcPswd()
{
    PrivateData* data = (PrivateData*)malloc(sizeof(PrivateData));
    std::cin >> data->m_pswd;
    doSmth(*data);
    memset(data, 0, sizeof(PrivateData));
    free(data);
    return 1;
}

int main()
{
    funcPswd();
    return 0;
}
```

我们在发布版本中测试，由于调试版本含有所有的调用在我们希望它们调用的时候。在 Visual Studio 2015 编译后，我们得到以下汇编代码：

```
....
000000013FBB1021    mov             rcx,
            qword ptr [__imp_std::cin (013FBB30D8h)]
```

```
000000013FBB1028    mov             rbx,rax
000000013FBB102B    lea             rdx,[rax+8]
000000013FBB102F    call
            std::operator>><char,std::char_traits<char> >
(013FBB18B0h)
000000013FBB1034    mov             rcx,rbx
000000013FBB1037    call            doSmth (013FBB1170h)
000000013FBB103C    xor             edx,edx
000000013FBB103E    mov             rcx,rbx
000000013FBB1041    lea             r8d,[rdx+70h]
000000013FBB1045    call            memset (013FBB2A2Eh)
000000013FBB104A    mov             rcx,rbx
000000013FBB104D    call            qword ptr [__imp_free
(013FBB3170h)]
        return 0;
000000013FBB1053    xor             eax,eax
....
```

Visual Studio 这次做得很好 :它按预期清理了数据。但是其他的编译器呢？让我们尝试 gcc 版本 5.2.1，和 clang，版本 3.7.0。

我修改一下代码来适应 gcc 和 clang，添加一些代码打印输出清理之前和之后分配内存块的内容。在内存释放后，将指针指向内存块的内容输出，但在实

际的程序中不能这样做，因为你永远不知道应用程序将如何响应。然而，在这个实验中，我可以自由地使用这种技术。

```
....
#include "string.h"
....
size_t len = strlen(data->m_pswd);
for (int i = 0; i < len; ++i)
   printf("%c", data->m_pswd[i]);
printf("| %zu \n", data->m_hash);
memset(data, 0, sizeof(PrivateData));
free(data);
for (int i = 0; i < len; ++i)
   printf("%c", data->m_pswd[i]);
printf("| %zu \n", data->m_hash);
....
```

现在，这里有一段由编译器 gcc 生成的汇编代码：

```
movq (%r12), %rsi
movl $.LC2, %edi
xorl %eax, %eax
call printf
movq %r12, %rdi
call free
```

打印功能（printf）后继调用 free() 函数同时调用 memset() 函数的功能则消失了。如果我们运行代码，任意输入一个密码（例如“MyTopSecret”），我们会看到如下信息打印在屏幕上：

MyTopSecret| 7882334103340833743

MyTopSecret| 0

哈希值改变了。我猜想这是内存管理副作用。至于我们的密码“ MyTopSecret”，它在内存中保持完整。

让我们来看看它是如何在 clang 中工作的：

```
movq (%r14), %rsi
movl $.L.str.1, %edi
xorl %eax, %eax
callq printf
movq %r14, %rdi
callq free
```

正如在之前的例子一样，编译器决定删除 memset() 的调用。打印输出为：

MyTopSecret| 7882334103340833743

MyTopSecret| 0

所以，gcc 和 clang 决定优化我们的代码。因为调用 memset() 后释放内存，编译器认为这样不重要并删除它。

我们的实验表明，编译器会倾向删除 memset() 调用为了栈和应用程序的动态内存工作优化的缘故。最后，让我们看看在使用新的操作符分配内存时，编译器将如何响应。

再次修改代码：

```
#include <string>
#include <functional>
#include <iostream>
#include "string.h"

struct PrivateData
{
   size_t m_hash;
   char m_pswd[100];
};

void doSmth(PrivateData& data)
{
   std::string s(data.m_pswd);
   std::hash<std::string> hash_fn;

   data.m_hash = hash_fn(s);
}

int funcPswd()
{
```

```
    PrivateData* data = new PrivateData();
    std::cin >> data->m_pswd;
    doSmth(*data);
    memset(data, 0, sizeof(PrivateData));
    delete data;
    return 1;
}

int main()
{
    funcPswd();
    return 0;
}
```

Visual Studio 中清理内存，正如预期一致：

```
000000013FEB1044  call     doSmth (013FEB1180h)
000000013FEB1049  xor      edx,edx
000000013FEB104B  mov      rcx,rbx
000000013FEB104E  lea      r8d,[rdx+70h]
000000013FEB1052  call     memset (013FEB2A3Eh)
000000013FEB1057  mov      edx,70h
000000013FEB105C  mov      rcx,rbx
000000013FEB105F  call     operator delete (013FEB1BA8h)
```

return 0;

000000013FEB1064 xor eax,eax

gcc 编译器同样决定省去清理函数：

call printf

movq %r13, %rdi

movq %rbp, %rcx

xorl %eax, %eax

andq $-8, %rdi

movq $0, 0(%rbp)

movq $0, 104(%rbp)

subq %rdi, %rcx

addl $112, %ecx

shrl $3, %ecx

rep stosq

movq %rbp, %rdi

call _ZdlPv

打印输出已变化，我们输入的数据已不存在：

MyTopSecret| 7882334103340833743

| 0

至于 clang，在这种情况下，它选择减掉“不必要”的函数来优化我们的代码：

movq (%r14), %rsi

```
movl $.L.str.1, %edi
xorl %eax, %eax
callq printf
movq %r14, %rdi
callq _ZdlPv
```

打印输出内存中的内容：

```
MyTopSecret| 7882334103340833743
MyTopSecret| 0
```

密码仍然存在，等待被窃取。

让我们总结一下。我们发现，一个优化的编译器可以将删除 memset() 调用，无论使用什么类型的内存-堆栈或动态。虽然在使用动态存储测试中 Visual Studio 不删除 memset() 调用，你不能指望它在实际代码中总是那样。有害的影响可能会泄露自身与其他编译转换。接下来我们做一个小研究的，即能不能依靠 memset() 函数来清理隐私数据。

那么，什么方式可以更好的来清理他们？

你应该使用特殊的内存清理功能，在优化代码的时候，它不会被编译器删除。

在 Visual Studio 中，例如，你可以使用 RtlSecureZeroMemory。从 C11 开始，memset_s 函数也可以使用。此外，如果需要的话，你可以实现自己的安全功能，网上可以找到许多实例和指南。这里列举出一些。

解决方案 No. 1。

```
errno_t memset_s(void *v, rsize_t smax, int c, rsize_t n) {
   if (v == NULL) return EINVAL;
```

```
  if (smax > RSIZE_MAX) return EINVAL;
  if (n > smax) return EINVAL;
  volatile unsigned char *p = v;
  while (smax-- && n--) {
    *p++ = c;
  }
  return 0;
}
```

Solution No. 2.

```
void secure_zero(void *s, size_t n)
{
    volatile char *p = s;
    while (n--) *p++ = 0;
}
```

一些程序员甚至进一步创建函数，填充数组中的伪随机值，并有不同的运行时间，以阻止基于实际测量的攻击。实现方法同样可以在网络中搜索到。

结束语

PVS Studio 静态分析器可以检测我们这里讨论的数据清理错误，并使用有关该问题的 诊断程序 V597。这篇文章作为一个扩展的解释，讲述为什么这个诊断是重要的。不幸的是，许多程序员倾向于认为分析器“选择”他们的代码，

实际上没有什么可以担心的。嗯，那是因为他们在代码调试器中看到自己的 memset() 调用完整，而忘记了他们所看到的仍然只是一个调试版本。