An Analysis of Address Space Layout Randomization on Windows Vista

Ollie Whitehouse, *Architect, Symantec Advanced Threat Research*

***Abstract*: Address Space Layout Randomization (ASLR) is a prophylactic security technology aimed at reducing the effectiveness of exploit attempts. With the advent of Microsoft Windows Vista, ASLR has been integrated into the default configuration of a Microsoft Windows operating system for the first time. We measure the behavior of the ASLR implementation in the Vista RTM release. Our analysis of the results uncovers predictability in the implementation that reduces its effectiveness.**

***Index Terms*: Address Space Layout Randomization, ASLR, Security, Microsoft, Windows, Vista, Visual Studio, /Dynamicbase**

I. Introduction

Address space layout randomization, or ASLR, is a prophylactic security technology that increases system security by increasing the diversity of attack targets [20]. Rather than increasing security by removing vulnerabilities from the system, ASLR makes it more difficult to exploit existing vulnerabilities. This technology is complementary to efforts to remove security vulnerabilities since it can offer some protection for vulnerabilities that are not yet known or have not yet been remedied. ASLR is also complementary to other prophylactic techniques such as Data Execution Prevention (DEP) since the combination of both technologies provides a much stronger defense against *memory manipulation vulnerabilities* than either one alone.

Techniques for exploiting *memory manipulation vulnerabilities* are sensitive to the memory layout of the program being targeted. This important class of vulnerability includes stack and heap overflows, underflows, format string vulnerabilities, array index overflows and uninitialized variables. By randomizing the memory layout of an executing program, ASLR decreases the predictability of the memory layout and reduces the probability that an individual exploit attempt will succeed. The security offered by ASLR is based on several factors [12], including how predictable the random memory layout of a program is, how tolerant an exploit technique is to variations in memory layout, and how many exploitation attempts an attacker can practically make.

Address space layout randomization has been integrated into and available for several popular operating systems such as OpenBSD and Linux for several years. Third-party ASLR implementations have been available for previous versions of the Microsoft Windows operating systems as stand-alone products or as part of a Host intrusion protection (HIPS) solutions [1][2][4][11][18]. With the advent of Microsoft Windows Vista, ASLR has been integrated into the default configuration of a Microsoft Windows operating system for the first time [5].

We measured the behavior of the address space layout randomization feature in the 32-bit Vista RTM release. This paper presents our measurements and discusses our measurement techniques. Our analysis uncovers some flaws that reduce the effectiveness of Vista's ASLR implementation.

A. Previous Work

There have been several efforts to describe and measure the ASLR implementation in Vista. The first is described in the tuxedo-es.org blog [13] and supported by the release of their VistaProbe tool [14][15][16][17]. This was followed shortly by a paper by Rahbar [10] but it's analysis was refuted by Howard [6]. Both efforts used a beta release of Vista to make their measurements since a release version was not yet available.

To our knowledge our work is the first to measure the ASLR implementation in the Vista RTM release. Unlike previous work, we rebooted the test system between measurements to measure ASLR in the environment it was intended to be used in. While previous work relied on a small number of measurements to draw conclusions, we took a much larger number of measurements and expect our results to have more statistical significance. Finally, we report several important and unexpected deficiencies in the ASLR implementation that were not previously reported and have since been acknowledged by Microsoft.

B. Outline

The remainder of this paper is organized as follows: Section II describes the ASLR implementation provided by

Vista and describes our methodology for measuring its behavior. Section III presents and analyzes the measurements we made. Section IV presents our conclusions based on the analysis.

II. ASLR IN VISTA

Windows Vista provides address space layout randomization on a per-image basis. Any executable image which contains a PE header, such as executable binaries (.exe) and dynamic link libraries (.dll), can elect to participate in address space layout randomization. This election is made by setting a bit (0x40) in one of the PE header fields (DLLCHARACTERISTICS) [7]. An option (/dynamicbase) is provided by the Microsoft Visual Studio 2005 linker for setting this bit when linking an image.

While loading an image that has elected to participate in ASLR the system uses a random global image offset. This offset is selected once per reboot, although we've uncovered at least one other way to cause this offset to be reset without a reboot (see Appendix II). The image offset is selected from a range of 256 values and is 64kB aligned. This offset and the other random parameters are generated pseudo-randomly [3]. All images loaded together into a process, including the main executable and DLLs, are loaded one after another at this offset. Because image offsets are constant across all processes a DLL that is shared between processes can be loaded at the same address in all processes for efficiency.

When executing a program whose image has been marked for ASLR the memory layout of the process is further randomized by randomly placing the thread stack and the process heaps. The stack address is selected first. The stack region is selected from a range of 32 possible locations, each separated by 64kB or 256kB (depending on the STACK_SIZE setting). Once the stack has been placed, it the initial stack pointer is further randomized by decrementing it a random amount. The initial offset is selected to be up to half a page (2048 bytes) but is limited to naturally aligned addresses (four byte alignment on IA32 and 16 byte alignment on IA64). The choices result in an initial stack pointer chosen from one of 16384 possible values on an IA32 system.

After the stack address has been selected, the process heaps are selected. Each heap is allocated from a range of 32 different locations, each separated by 64kB. The location of the first heap must be chosen to avoid the previously placed stack, and each of the following heaps must be allocated to avoid those allocated before it.

The address of an operating system structure known as the *Process Environment Block* (PEB) is also selected randomly. The PEB randomization feature was introduced earlier in Windows XP SP2 and Windows 2003 SP1 and is also present in Windows Vista. Although implemented separately it is also a form of address space randomization but unlike the other ASLR features, PEB randomization occurs whether or not the executable being loaded elected to use the ASLR feature.

An important result of Vista's ASLR design is that some address space layout parameters such as PEB, stack and heap locations are selected once per program execution. Other parameters, such as the location of the program code, data segment, BSS segment and libraries, change only between reboots.

A. Methodology

We measured Vista's ASLR implementation to verify its behavior and to determine how random the memory layout of loaded programs is. We constructed a program to log several important addresses associated with the program each time it is executed. To measure the randomization of the image base address the test program prints the address of a function in the code segment. To measure the randomization of the stack, it prints the address of an automatic variable. To measure the randomization of the PEB structure it prints the address of the PEB. Finally it measures the placement of three heaps. It measures the CRT heap by printing the first value returned by malloc. It prints the first values returned by HeapAlloc using the default process heap, and using a heap created with HeapCreate.

We compiled our test program with Microsoft Visual Studio 2005 SP1 Beta and linked it statically. The program source is listed in Appendix I. When executed repeatedly with the same environment on a machine that does not support ASLR or PEB randomization, this program reports the same constants each time. Any variation in the output is due solely to the effects of ASLR.

Some ASLR parameters are only set once per system boot[1]. To properly measure the effects of ASLR we decided to reboot the system between measurements. We configured an AMD Ahtlon 3200 system running 32-bit Windows Vista RTM to automatically log in and run our data collection utility during system startup and then to reboot. This setup closely mimics the environment of long running servers which are executed once during system startup. Our test harness was used to collect samples from 11,500 test runs over the course of twelve days. The complete data is being made available [19].

III. ANALYSIS

We analyzed the results of our measurements to quantify the amount of randomness introduced into the memory layout of a process by the ASLR implementation. Figure 1 show the collected data for HeapAlloc addresses plotted as a series of samples. The plot reveals no noticeable patterns, indicating some amount of randomness. Plots for the other measured addresses were similar and are not included here.

[1] Normally; as noted previously.

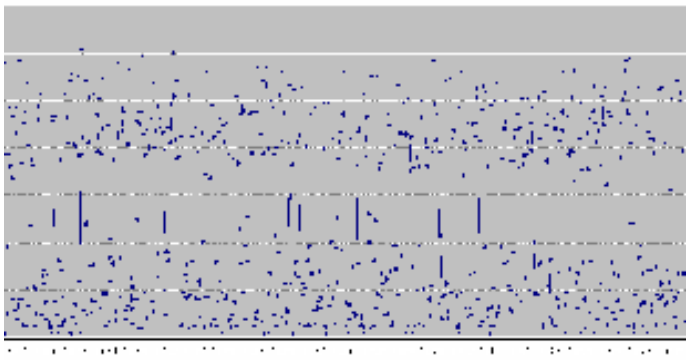


Figure 1 – 11,500 HeapAlloc samples.

To compare the implementation design with our measurements we analyzed the data to determine how many different value each parameter can have. Table 1 shows a count of unique values found in the 11,500 samples taken for each measurement and compares this count to the expected range dictated by the design.

The observed range of stack addresses differs from what is expected but it is clear that our 11,500 samples are not large enough to properly measure the full range and no conclusions can be drawn.

Item	Expected	Observed	Difference
Stack	16384 (2^{14})	8,568	-48%
`Malloc`[2]	>= 32 (>= 2^5)	192	+500%
HeapAlloc[3]	>= 32 (>= 2^5)	95	+200%
CreateHeap[4]	>= 32 (>= 2^5)	209	+550%
Image	256 (2^8)	255	-0.4%
PEB	16 (2^4)	13	-19%

Table 1 – Comparison between the number of unique values observed and expected in each data set.

The observed range of heap addresses differs considerably from each other and from their expected values. All three heap allocations show more variability than expected. This may indicate that our description of heap address selection is incomplete or incorrect. Another possible explanation is that heap use during program startup is non-deterministic and adds to the randomness of our samples. One surprising result is that the observed range of HeapAlloc locations is much smaller than the range of addresses allocated using malloc! In 11,500 runs we observed 95 unique addresses were returned by `HeapAlloc()` while 192 unique addresses were returned by `malloc()`. The difference in usage reveals that applications that utilize the Microsoft `HeapAlloc()` function are at greater risk than those that utilize the ANSI C `malloc()` API. This could be due to the order in which heaps are allocated. A heap that is created later will have to be placed at an address not occupied by previously allocated heaps, reducing the amount of randomness in its placement.

A. Occurrences of duplicates

The range of a random variable does not tell the whole story. The protection offered by ASLR depends on the entropy of the parameters, that is, how unpredictable they are. The entropy of a parameter with a given range is highest (most unpredictable) when all values are equally likely.

We observed a significant number of instances where the same address was returned for a parameter in successive test runs. We compared the occurrences of these duplicates with the amount of duplicates we would expect if all sample values were equally likely. This comparison is shown in Table 2. Again, due to the number of samples no meaningful conclusions can be drawn from the stack measurements.

Item	Expected	Observed	Difference
Stack	< 1 / < 1	1	0% / 0%
Malloc	359 / 60	133	-63% / +120%
HeapAlloc	359 / 121	176	-51% / +45%
CreateHeap	359 / 55	130	-64% / +140%
Image	45 / 45	39	-13% / -13%
PEB	719 / 884	1,322	+84% / +50%

Table 2 – Comparison between the number of successive duplicates expected and observed in each data set.
Expected values show how many duplicates are expected in 11,500 samples based on the theoretical range and based on the observed range.

The occurrences of duplicates in heap and PEB addresses deviate significantly from what is expected. Of 11,500 HeapAlloc address samples, successive samples reported the same value 176 times or 1.5% of the time. If all 32 expected values were equally likely, the probability of successive updates would be 3.1% and there would be about 359 duplicates, 51% more than were observed. If all 95 observed values were equally likely the probability of successive duplicates would be 1.1% and there would be only about 121 successive duplicates, 45% less than were observed. This discrepancy suggests that the 95 observed values are probably not uniformly distributed and there may be some bias in the selection of heap addresses when using HeapAlloc. To determine if this discrepancy was due to an obvious pattern, we plotted the number of runs between duplicate values to see if they yielded any obvious pattern, but did not find any (see Appendix IV).

The deviation between expected and observed is even larger for duplicates in the malloc and PEB samples. Duplicates occurred 1.2 times as often as expected in heap

[2] Heap allocation using the malloc function.
[3] Heap allocation using the HeapAlloc function and the default process heap.
[4] Heap allocation using HeapAlloc and a heap created with the CreateHeap function.

addresses returned by malloc where 133 duplicates account for 1.2% of the samples rather than the 60 (0.5%) expected if all 192 observed values were equally likely. Similarly 1,322 duplicates account for 11% of the PEB address samples rather than the mere 719 (6.3%) expected if there were 16 equally likely addresses. These discrepancies point to even larger biases in the randomization of malloc and PEB addresses.

Duplicates occurred 13% less often than expected in the image addresses. This also suggests the presence of bias, although the deviation is smaller than in the other instances.

B. Frequency Distribution Analysis

A distribution that has biases that favor some values over other values is more predictable and has lower entropy. We plotted the distributions of the samples for each parameter we measured to identify any biases.

The distribution of the stack address is shown in Figure 2. The graph shows a near-uniform distribution with no significant biases. Combined with the large range this indicates that the stack addresses should be fairly unpredictable.

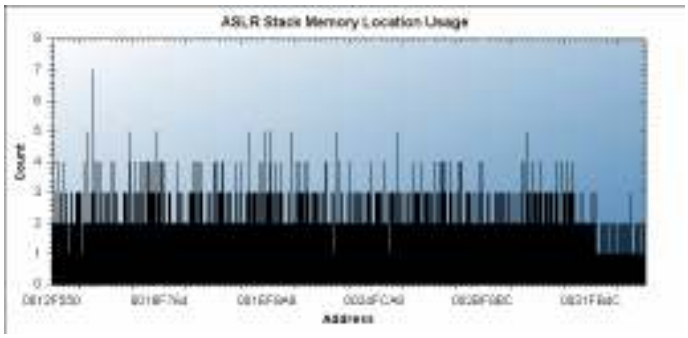
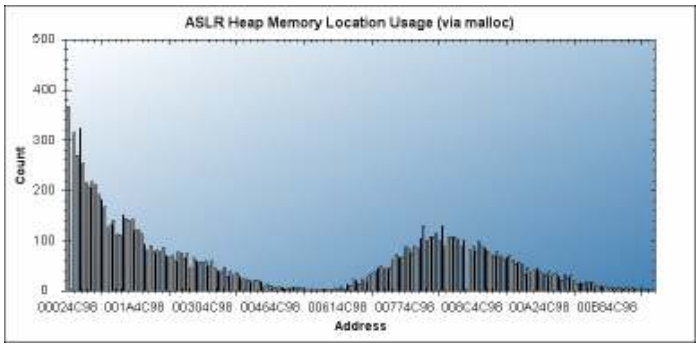
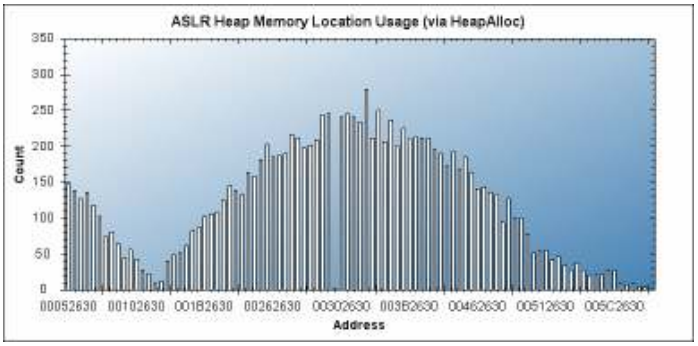


Figure 2 – Distribution of stack addresses.

Significant biases are apparent in the distribution of heap addresses as seen in Figure 3. These parameters also have a much smaller range. As seen in Figure 4, there is also bias in the distribution of code addresses (which are affected by randomization of the image base address) although these biases are not as profound as those seen in the heap distributions.





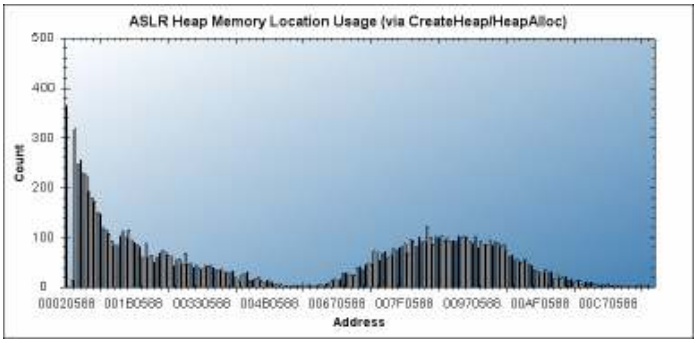


Figure 3 – Distribution of heap addresses using malloc, HeapAlloc and HeapAlloc with CreateHeap.

Finally, PEB randomization shows a significant bias to one address. Figure 5 shows that 25% of all executions chose a single address and another 10% of all executions chose a second address. These two addresses were chosen 35% of the time and the remaining eleven addresses were chosen only 65% of the time.

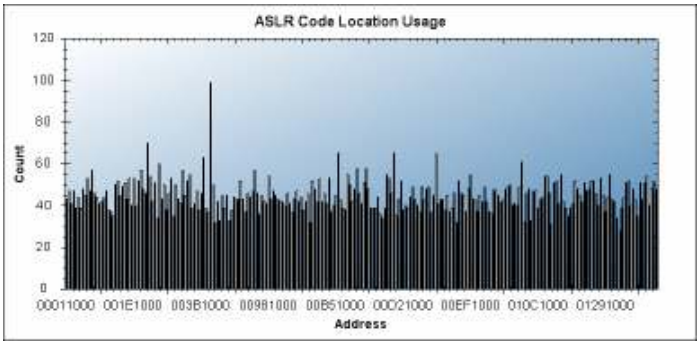


Figure 4 – Distribution of code addresses.

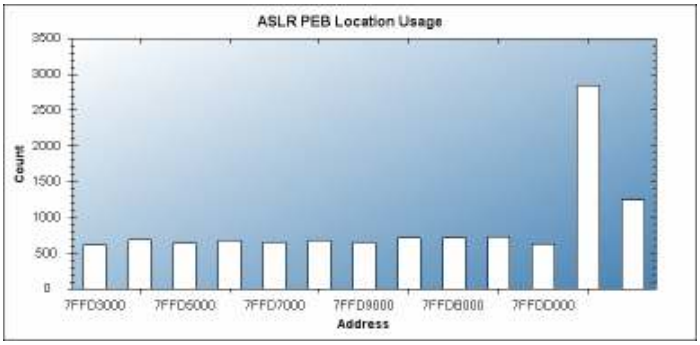


Figure 5 – Distribution of PEB addresses.

Because the distributions of image base, heap and PEB addresses have biases, they are more predictable than their range measurements alone would indicate. For example, if all 16 PEB addresses were equally likely, an attacker would have a 6.25% chance of guessing the PEB address but because of biases an attacker can make a single guess that is correct 25% of the time.

IV. Conclusions

This paper shows that the stack, heap, image and PEB protected by ASLR on Microsoft Windows Vista 32bit RTM have different frequency distributions. While the stack has near uniform distribution over a very wide range, the heap and PEB, and to a lesser degree the image base have much smaller ranges and because of biases in their distributions do not efficiently use this range. As a result, the protection offered by ASLR under Windows Vista may not be as robust as expected. Microsoft has acknowledged the problems with PEB and image vase randomization and indicated that it is caused by a weakness in the implementation.

This paper also shows that applications that leverage Microsoft's `HeapAlloc()` function are not afforded the same level of protection as those that leverage the ANSI C heap allocation API `malloc()`. The impact of which is that third party software that explicitly use Microsoft's API, are potentially more vulnerable from successful exploitation than software that does not. Also apparent is that using `CreateHeap()` then `HeapAlloc()` improves the entropy slightly over solely using `malloc()`.

Finally, results show that there are less consecutive duplicates than expected in the PEB randomization. This result adds to the evidence that the source of entropy used within ASLR is poorly used.

Acknowledgment

The author would like to acknowledge the help and support of Oliver Friedrichs, Matt Conover, Zulfikar Ramzan, of Symantec and Tim Newsham of iSEC Partners. The author would also like to acknowledge Nitin Kumar Goel, of Microsoft who reviewed this research and provided candid feedback.

References

[1] E. Berger, DieHard, http://www.cs.umass.edu/~emery/diehard/.
[2] R. Brown, DieHarder, http://www.phy.duke.edu/~rgb/General/dieharder.php.
[3] I. Hellen, V. Kumar, "Security Engineering in Vista," Sep. 2006, http://packetstormsecurity.org/hitb06/DAY_1_-_Ian_Hellen_and_Vishal_Kumar_-_Security_Engineering_in_Vista.pdf.
[4] M. Howard, "Address Space Layout Randomization for Windows," Sep. 2005, http://blogs.msdn.com/michael_howard/archive/2005/09/30/475763.aspx.
[5] M. Howard, "Address Space Layout Randomization in Windows Vista," May 2006, http://blogs.msdn.com/michael_howard/archive/2006/05/26/608315.aspx.
[6] M. Howard, "Alleged Bugs in Windows Vista's ASLR Implementation," Oct. 2006, http://blogs.msdn.com/michael_howard/archive/2006/10/04/Alleged-Bugs-in-Windows-Vista_1920_s-ASLR-Implementation.aspx.
[7] Microsoft, "Microsoft Portable Executable and Common Object File Format Specification," May 2006, http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx.
[8] N. Goel, Microsoft, Private communications, Jan. 2007.
[9] Pageexec, "Protecting against Pointer Subterfuge (Redux)," Dailydave mailing list, Oct 2006, http://archives.neohapsis.com/archives/dailydave/2006-q4/0028.html
[10] A. Rahbar, "An analysis of Microsoft Windows Vista's ASLR," Oct. 2006, http://www.sysdream.com/articles/Analysis-of-Microsoft-Windows-Vista's-ASLR.pdf.
[11] Security Architects, Ozone, http://www.securityarchitects.com/products.html.
[12] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, D. Boneh, "On the Effectiveness of Address-Space Randomization," In *Proceedings of 11th ACM Conference on Computer and Communications Security*, Oct. 2004, http://www.stanford.edu/~blp/papers/asrandom.pdf.
[13] Tuxedo-es, "Microsoft Windows Vista: Measuring the security enhancements," Jun. 2006, http://www.tuxedo-es.org/blog/2006/06/11/microsoft-windows-vista-measuring-the-security-enhancements/.
[14] Tuxedo-es, "Microsoft Windows Vista beta-2 build 5384: Vista-Probe 0.1 results," Jun. 2006, http://www.tuxedo-es.org/blog/2006/06/13/microsoft-windows-vista-beta-2build-5384-vista-probe-01-results/.
[15] Tuxedo-es, "Vista Probe 0.1 release," Jun. 2006, http://www.tuxedo-es.org/blog/2006/06/15/vista-probe-01-released/.
[16] Tuxedo-es, "Vista Probe 0.2 release," Jul. 2006, http://www.tuxedo-es.org/blog/2006/07/06/vista-probe-02-release/.
[17] Tuxedo-es, "Vista Probe 0.2 test case sources," http://pearls.tuxedo-es.org/win32/vista-probe-0.2_tests_sources.zip.
[18] Wehnus, "Whentrust," http://www.wehntrust.com/.
[19] O. Whitehouse, "Vista address space randomization data," Feb 2007, Available via E-Mail
[20] Wikipedia, "Address Space Layout Randomization," http://en.wikipedia.org/wiki/Address_Layout_Randomization.

I. ASLR.CPP

The following code was used to collect the source data for the analysis of ASLR.

```
//
// (c)2006 Symantec Corporation
// Ollie Whitehouse - ollie_whitehouse@symantec.com
//

#include "stdafx.h"


// ------------------------------------------------------------------------
// Function: Banner
// Description:  Print banner
// ------------------------------------------------------------------------
void banner(){
  fprintf(stdout,"--------------------------------------------------------------------------------\n");
  fprintf(stdout,"ASLR - v0.1\n");
  fprintf(stdout,"(c)2006 Symantec Corporation\n");
  fprintf(stdout,"--------------------------------------------------------------------------------\n\n");
}

//
// Prints the location of the function in memory
// This is used to validate .code randomization
//
int verifyCode(FILE  *fileCSV)
{
  fprintf(stdout,"%0.8p\n",verifyCode);
  fprintf(fileCSV,"%0.8p\n",verifyCode);

  return 0;
}

//
// Some of this function taken from Phrack 62
// Prints the location of the PEB in memory
//
int verifyPEB(FILE *fileCSV)
{
  DWORD* dwPebBase = NULL;

  /* Return PEB address for current process
     address is located at FS:0x30 */
    __asm
    {
      push eax
      mov eax, FS:[0x30]
      mov [dwPebBase], eax
      pop eax
    }

  fprintf(stdout,"%0.8X,",(DWORD)dwPebBase);
  fprintf(fileCSV,"%0.8X,",(DWORD)dwPebBase);

  return 0;
}

//
// Prints the location of a new heap in memory
//
int verifyHeapviaHeapCreate(FILE *fileCSV)
{
  HANDLE  hHeap;
  ULONG_PTR *varFoo;

  hHeap=HeapCreate(NULL,1024,2048);
  if(hHeap==NULL){
    fprintf(stdout,"error,");
    return 1;
  } else {
    varFoo=(ULONG_PTR *) HeapAlloc(hHeap,0,100);
```

```
    if(varFoo==NULL){
      fprintf(stdout,"error,");
      return 1;
    } else {
      fprintf(stdout,"%0.8p,",varFoo);
      fprintf(fileCSV,"%0.8p,",varFoo);
      HeapFree(hHeap,0,varFoo);
      return 0;
    }
    HeapDestroy(hHeap);
  }

  return 0;
}

//
// Prints the location of the heap in memory
//
int verifyHeapviaHeapAlloc(FILE *fileCSV)
{
  ULONG_PTR *varFoo;

  varFoo=(ULONG_PTR *) HeapAlloc(GetProcessHeap(),0,100);

  if(varFoo==NULL){
    fprintf(stdout,"error,");
    return 1;
  } else {
    fprintf(stdout,"%0.8p,",varFoo);
    fprintf(fileCSV,"%0.8p,",varFoo);
    HeapFree(GetProcessHeap(),0,varFoo);
    return 0;
  }

  return 0;
}

//
// Prints the location of the heap in memory
//
int verifyHeapviaMalloc(FILE *fileCSV)
{
  char *varFoo;

  varFoo=(char *) malloc(100);

  if(varFoo==NULL){
    fprintf(stdout,"error,");
    return 1;
  } else {
    fprintf(stdout,"%0.8p,",varFoo);
    fprintf(fileCSV,"%0.8p,",varFoo);
    free(varFoo);
    return 0;
  }

  return 0;
}

//
// Prints the location of the stack in memory
//
int verifyStack(FILE *fileCSV)
{
  int intFoo=1;

  fprintf(stdout,"%0.8p,",&intFoo);
  fprintf(fileCSV,"%0.8p,",&intFoo);

  return 0;
}

int _tmain(int argc, _TCHAR* argv[])
{
  FILE*fileCSV;
  TCHAR strFileName[MAX_PATH];

  banner();

  if(argc==2){
```

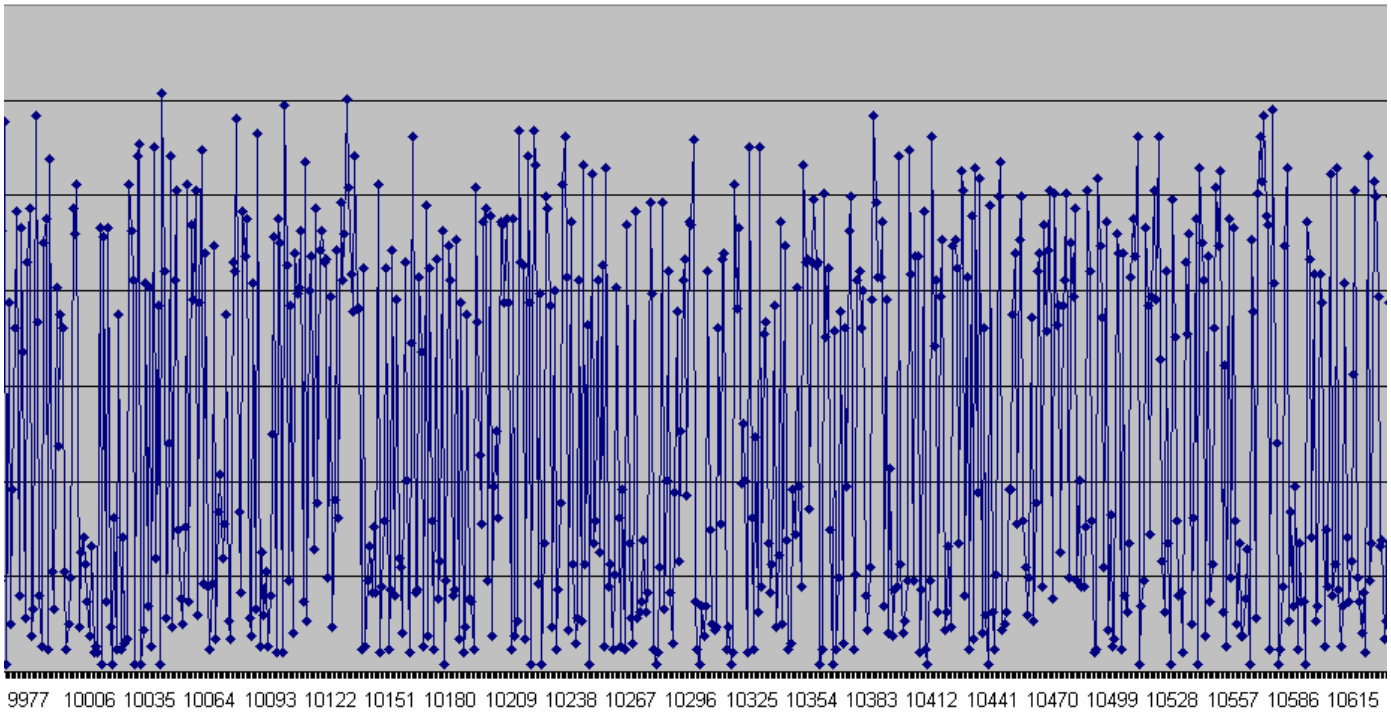
```
    _snwprintf_s(strFileName,MAX_PATH-1,_T("%s\\ASLR.csv"),argv[1]);
    //fwprintf(stdout,_T("%s\n"),strFileName);
    fileCSV=_wfopen(strFileName,_T("a"));
  } else {
    fileCSV=fopen("ASLR.csv","a");
  }

  if(fileCSV==NULL){
    fprintf(stdout,"[!] Couldn't open output file! Exiting!\n");
    return 1;
  }

  fprintf(stdout,"+-------+--------+--------+--------+--------+--------+\n");
  fprintf(stdout," Stack  | Heap 1 | Heap 2 | Heap 3 |  PEB   |  Code\n");
  fprintf(stdout,"+-------+--------+--------+--------+--------+--------+\n");

  verifyStack(fileCSV);
  verifyHeapviaMalloc(fileCSV);
  verifyHeapviaHeapAlloc(fileCSV);
  verifyHeapviaHeapCreate(fileCSV);
  verifyPEB(fileCSV);
  verifyCode(fileCSV);

  fclose(fileCSV);
  return 0;
}
```

II. RESEED.BAT

```
@echo off
REM (c) 2006 Symantec Corp - Ollie Whitehouse - ollie_whitehouse@symantec.com
REM This might seem a little strange but due to some odd instances
REM of not being able to delete/overwrite the executables (although the
REM original process should of finished) because of 'Access Denied'
REM this is the best way I found to do it without having to put an inefficient
REM 'sleep' in
:Reseed
Copy /Y ASLR.exe ASLR-1.exe
Copy /Y ASLR.exe ASLR-2.exe
Copy /Y ASLR.exe ASLR-3.exe
Copy /Y ASLR.exe ASLR-4.exe
Copy /Y ASLR.exe ASLR-5.exe
Copy /Y ASLR.exe ASLR-6.exe
Copy /Y ASLR.exe ASLR-7.exe
Copy /Y ASLR.exe ASLR-8.exe
Copy /Y ASLR.exe ASLR-9.exe
Copy /Y ASLR.exe ASLR-A.exe
Copy /Y ASLR.exe ASLR-B.exe
Copy /Y ASLR.exe ASLR-C.exe
Copy /Y ASLR.exe ASLR-D.exe
Copy /Y ASLR.exe ASLR-E.exe
Copy /Y ASLR.exe ASLR-F.exe
ASLR-1.exe
ASLR-2.exe
ASLR-3.exe
ASLR-4.exe
ASLR-5.exe
ASLR-6.exe
ASLR-7.exe
ASLR-8.exe
ASLR-9.exe
ASLR-A.exe
ASLR-B.exe
ASLR-C.exe
ASLR-D.exe
ASLR-E.exe
ASLR-F.exe
Del ASLR-1.exe
Del ASLR-2.exe
Del ASLR-3.exe
Del ASLR-4.exe
Del ASLR-5.exe
Del ASLR-6.exe
Del ASLR-7.exe
Del ASLR-8.exe
Del ASLR-9.exe
Del ASLR-A.exe
Del ASLR-B.exe
Del ASLR-C.exe
Del ASLR-D.exe
Del ASLR-E.exe
Del ASLR-F.exe
Goto Reseed
```

III. Heap Address Selection

The following graph sample shows for heap ASLR the difference in the addresses it selects for each run. The X axis is the run number and the Y axis is the address selected.

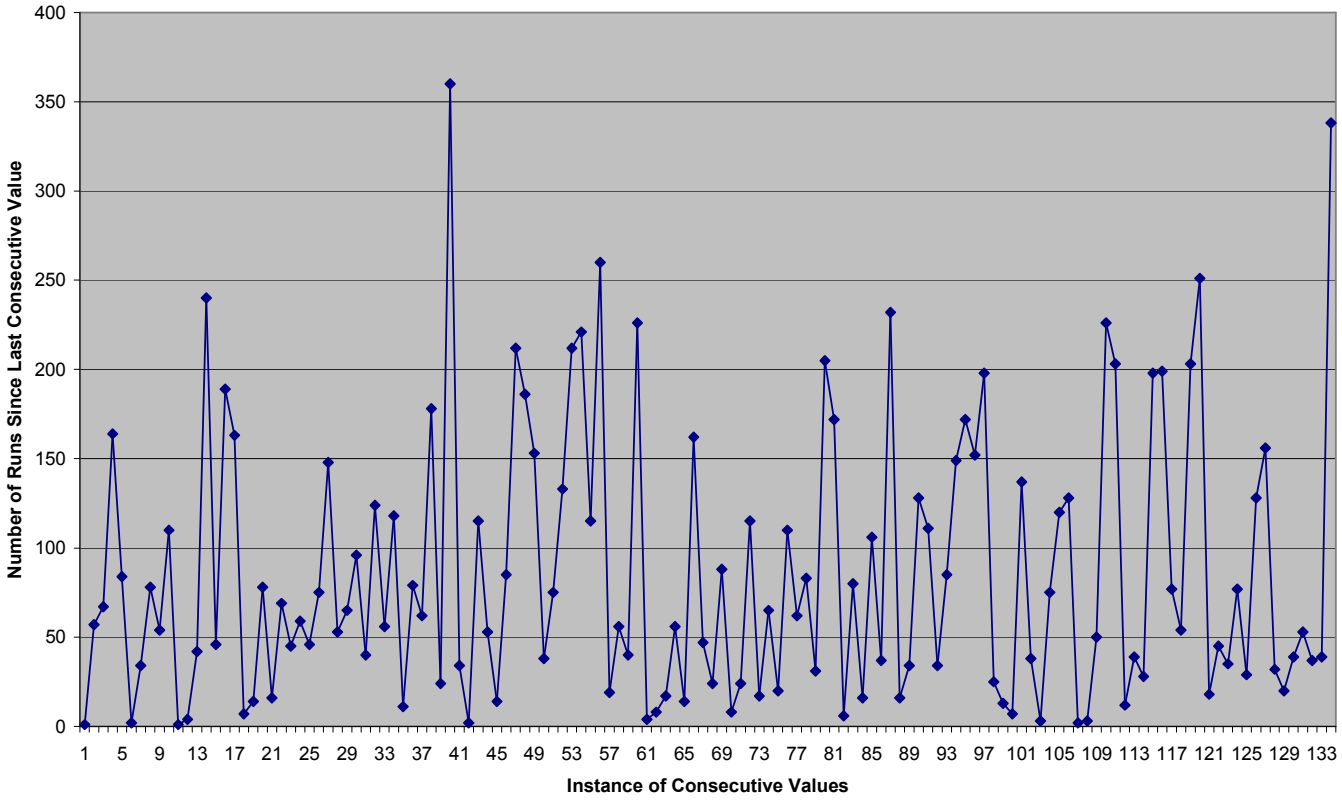


We can see from this small sample that there is no apparent pattern in address selection. The above is also true for the complete sample.

IV. Runs between Consecutive Values

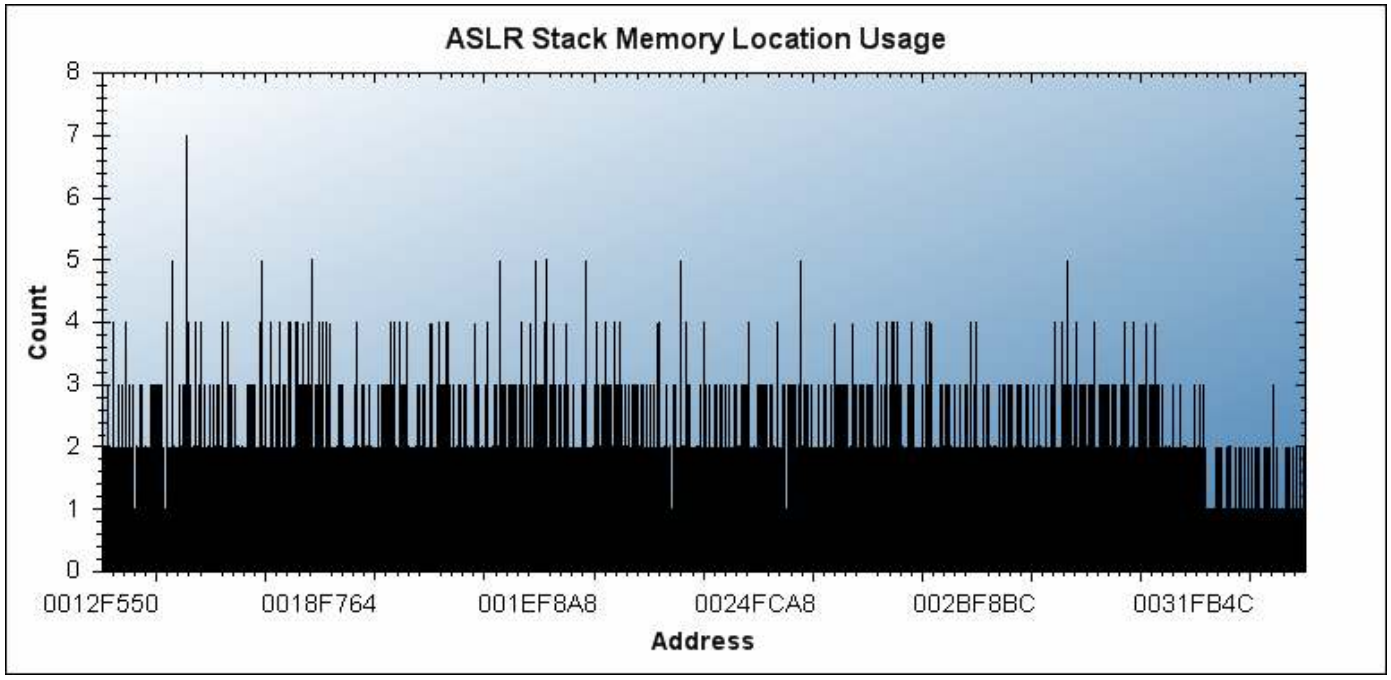
The following graph shows the 134 instances where adjacent runs resulted in the same memory address from the `HeapAlloc()` function. With a uniform distribution over a space with 2^8 (=256) elements, the probability that two consecutive memory addresses are the same is 1/256. Based on the results produced by the author we can see the Vista implementation is actually 134 from 11,500, which results in a likelihood of 1 in 85. This result is significantly higher than expected.

The number of runs between these 134 instances were then plotted, the result of which can be seen below. We can see that there was no obvious pattern which reduces the likelihood of an attacker accurately predicting the number of reboots required before the same memory address would be used consecutively.



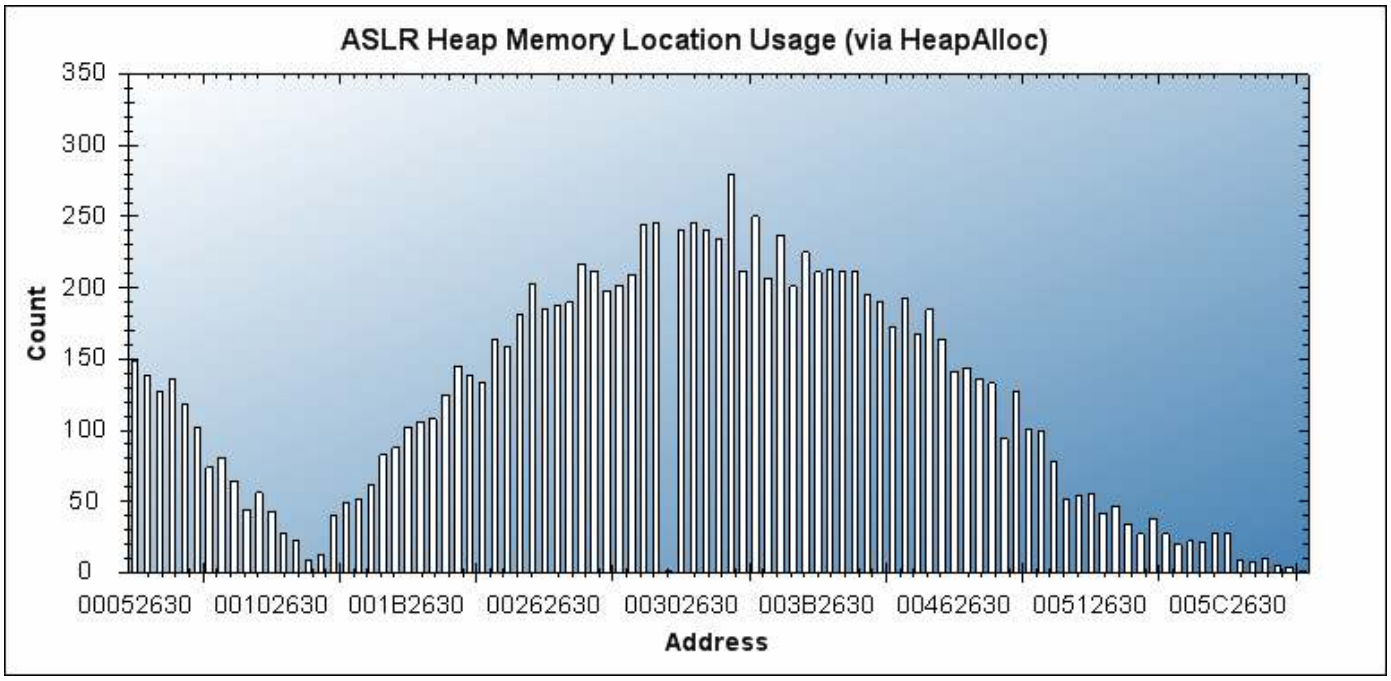
V. Distribution Graph - Stack

The following graph shows the frequency distribution of values for each of the stack locations observed during the execution of 11,500 instances.



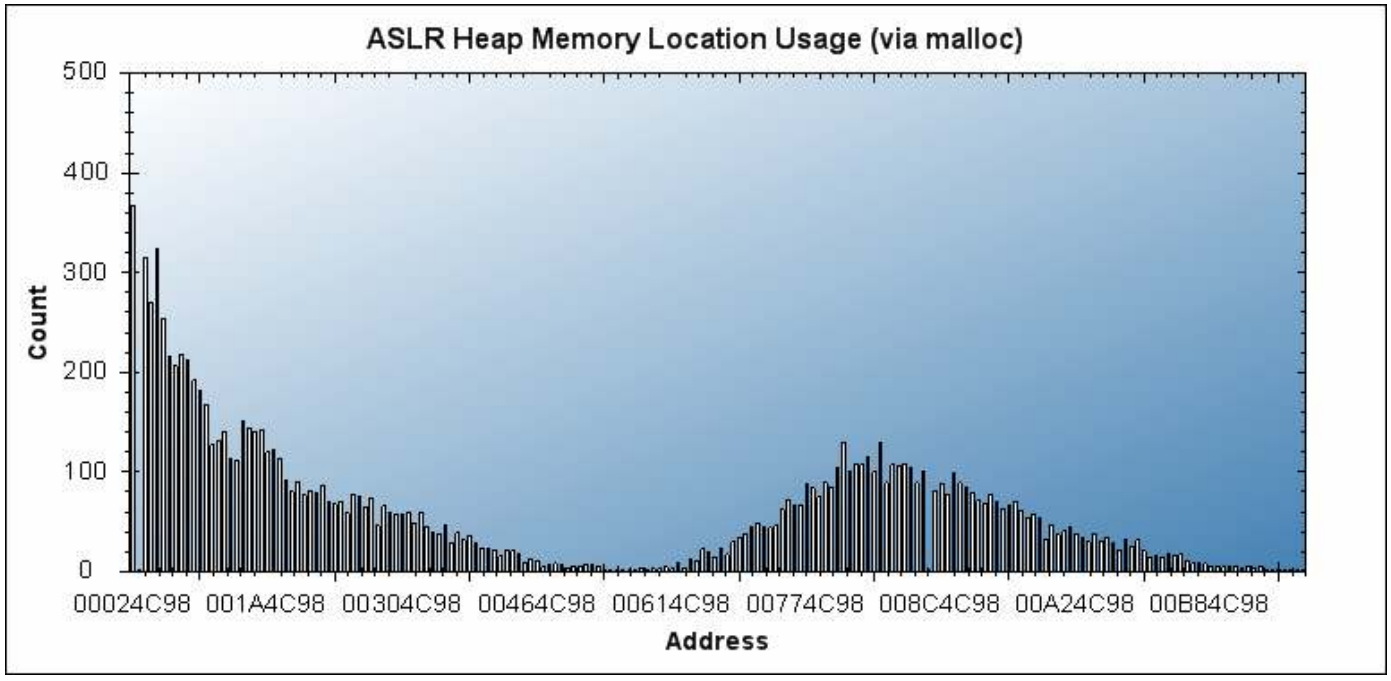
VI. Distribution Graph – Heap (via HeapAlloc function)

The following graph shows the frequency distribution of values for each of the heap locations observed during the execution of 11,500 instances. These were used via the `HeapAlloc()` function, which is a Windows specific API function.



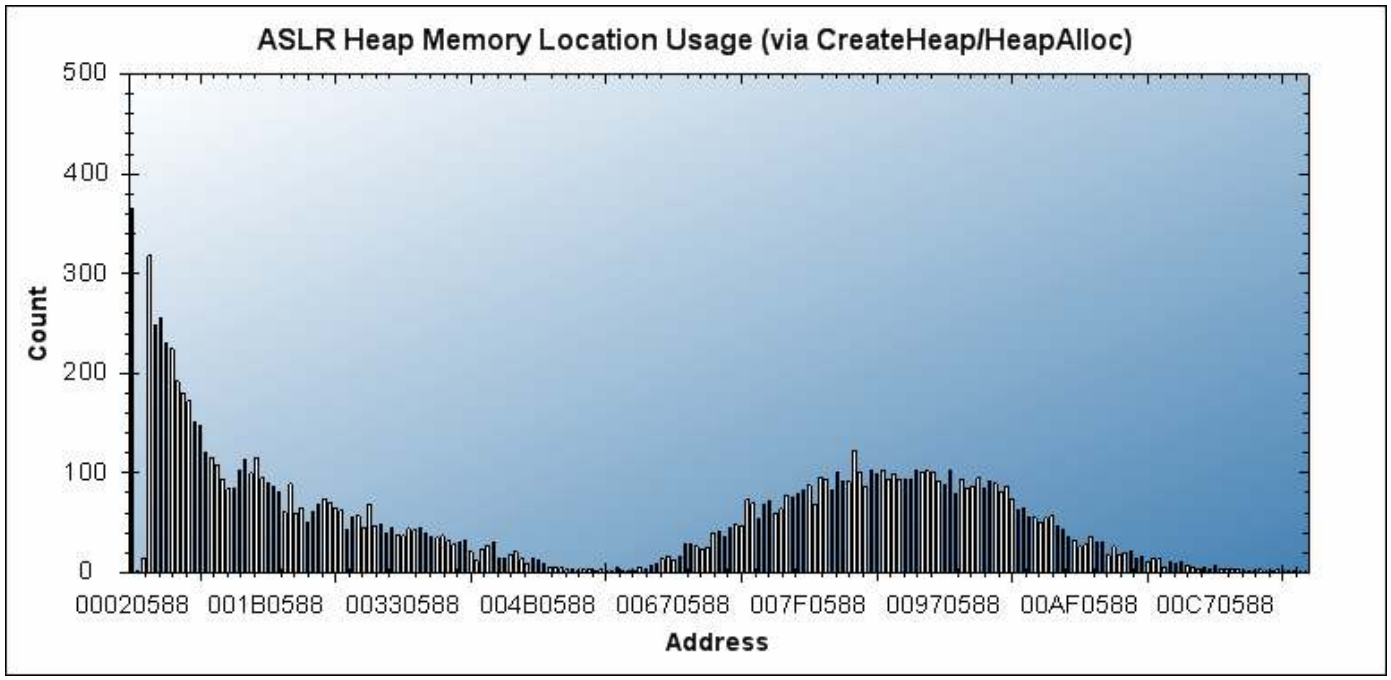
VII. Distribution Graph – Heap (via malloc function)

The following graph shows the frequency distribution of values for each of the heap locations observed during the execution of 11,500 instances. These were used via the `malloc()` function, which is a ANSI C API.



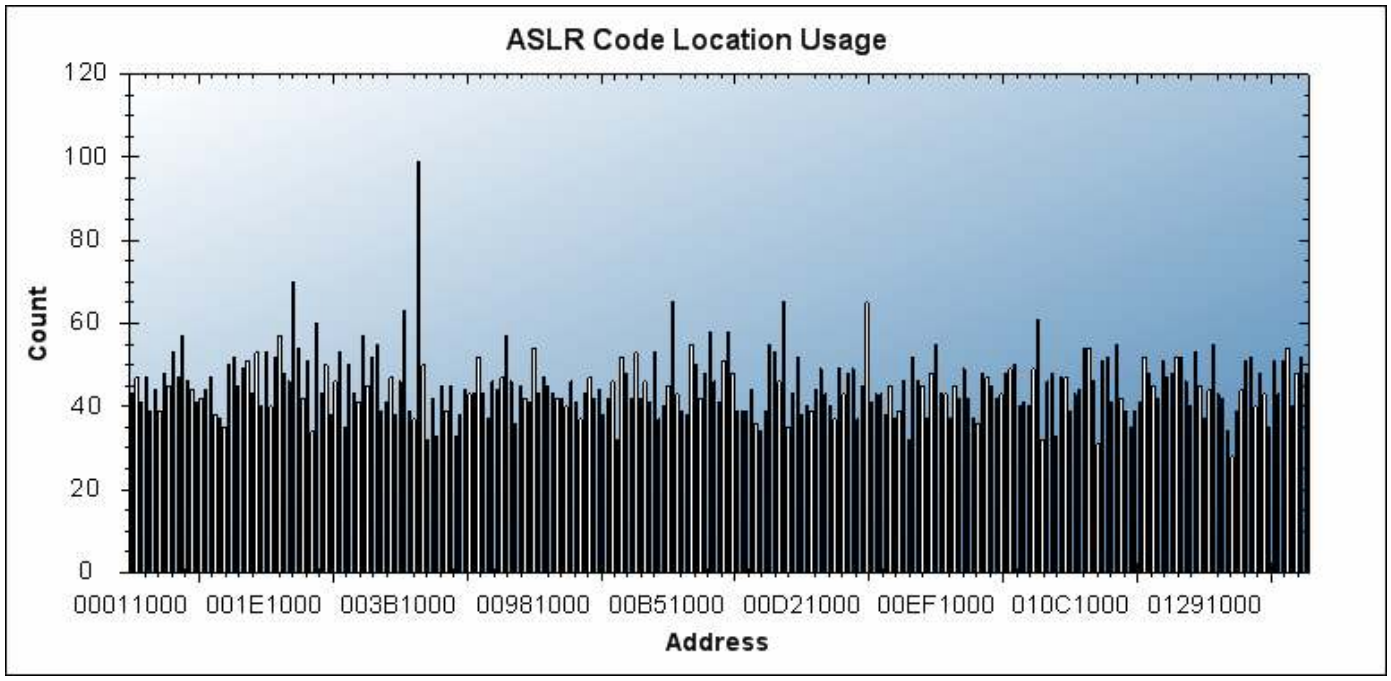
VIII. Distribution Graph – Heap (via CreatHeap and HeapAlloc functions)

The following graph shows the frequency distribution of values for each of the heap locations observed during the execution of 11,500 instances. These were used via the `CreateHeap()` and the `HeapAlloc()` functions, which are Windows specific APIs.



IX. Distribution Graphs - Image

The following graph shows the frequency distribution of values for each of the image locations observed during the execution of 11,500 instances.



X. Distribution Graphs –Process Environment Block

The following graph shows the frequency distribution of values for each of the PEB locations observed during the execution of 11,500 instances.