システム工学概論

7. Cにおけるセキュアプログラミング

山田泰司

taiji@aihara.co.jp

株式会社あいはら 研究開発チーム

脆弱なプログラムとは

ユーザから、なんらかの入力を受けとる際に：

- 悪意あるユーザにより、任意のコードが実行される
- 悪意あるユーザにより、制御の流れが変えられる
- ユーザにより、プログラムが意図せず終了する、
  もしくは意図せず終了しない
- ユーザに、意図せずローカルの情報が漏洩する
- 悪意あるユーザに、他のユーザが侵害される

攻撃の種類

- コードインジェクション
  （スタック破壊・ヒープ破壊など）
- アークインジェクション
  （ポインタ偽装・スタック破壊など）
- サービス拒否（整数オーバーフローなど）
- スパイ、盗難などのセキュリティ侵害
  （ディレクトリトラバーサル、不適切なメモリ管理、
  権限の昇格、悪意あるプロセスの常駐など）
- クロスサイトスクリプティングなど
  （不適切なウェブ、コンテンツサーバ管理など）

標準入力における脆弱性

脆弱なコード：

```
#include <stdio.h>

int main(void)
{
  char buf[64];

  gets(buf);
  puts(buf);
  return 0;
}
```

適切な標準入力

脆弱性への対処：secure000.c

```
#include <stdio.h>

int main(void)
{
  char buf[64];

  fgets(buf, sizeof(buf), stdin); /* gets は使うな！ */
  puts(buf);
  return 0;
}
```

適切な標準入力

脆弱性への対処：secure000.c

```
#include <stdio.h>

int main(void)
{
  char buf[64];

  gets_s(buf, sizeof(buf));
  puts(buf);
  return 0;
}
```

文字列の複製と結合における脆弱性

脆弱なコード：

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char buf[64] = "";

  if (argc == 2+1) {
    strcpy(buf, argv[1]);
    strcat(buf, "/");
    strcat(buf, argv[2]);
  }
  puts(buf);
  return 0;
}
```

適切な文字列の複製と結合

脆弱性への対処：secure001.c

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char buf[64] = "";

  if (argc == 2+1) { /* strlcpy, strlcat を使おう！ */
    strlcpy(buf, argv[1], sizeof(buf));
    strlcat(buf, "/", sizeof(buf));
    strlcat(buf, argv[2], sizeof(buf));
  }
  puts(buf);
  return 0;
}
```

適切な文字列の複製と結合

脆弱性への対処：secure002.c

```
#include <stdio.h>
#include <string.h>

/* strlcpy が無い場合は： */
#define strlcpy(d, s, z) do {\
  strncpy(d, s, z);\
  if ((z) > 0)\
    d[(z)-1] = '\0';\
} while (0)

/* strlcat が無い場合は： */
#define strlcat(d, s, z) strncat(d, s, (z)-strlen(d)-1)

int main(int argc, char *argv[])
{
  :
} /* Cにおける文字列のヌル終端に注意する */
```

適切な文字列の複製と結合

潜在的な脆弱性への対処：secure002.c

```
#include <stdio.h>
#include <string.h>

/* strlcpy が無い場合は： */
#define strlcpy(d, s, z) do {\
  strncpy(d, s, z);\
  if ((z) > 0)\
    d[(z)-1] = '\0';\
} while (0)

/* strlcat が無い場合は： */
#define strlcat(d, s, z) do { /* より安全な対処 */\
  if ((z) > strlen(d))                    \
    strncat(d, s, (z)-strlen(d)-1);       \
} while (0)

int main(int argc, char *argv[])
{
  :
} /* Cにおける文字列のヌル終端に注意する */
```

適切な文字列の複製と結合

脆弱ではない strcpy, strcat **の用法：**secure003.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  char *buf = NULL;

  if (argc == 2+1) {
    buf = malloc(strlen(argv[1]) + strlen(argv[2] + 1));
    if (buf) {
      strcpy(buf, argv[1]);
      strcat(buf, "/");
      strcat(buf, argv[2]);
    }
  }
  puts(buf);
  return 0;
}
```

オフバイワンエラー

脆弱なコード：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  int i;
#define LEN 16
  char str[LEN], *buf = NULL;

  strcpy(str, "0123456789abcdef");
  buf = malloc(strlen(str));
  if (buf) {
    for (i=1; i<=LEN; i++)
      buf[i] = str[i];
    buf[i] = '\0';
  }
  puts(buf);
  for (i=0; i<=argc; i++)
    puts(argv[i]);
  return 0;
}
```

適切なオフセット

脆弱性への対処：secure004.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{ /* うっかり全部、1 オフセットずれてた… */
  int i;
  char str[16], *buf = NULL;

  strncpy(str, "0123456789abcdef", sizeof(str)-1);
  str[sizeof(str)-1] = '\0';
  buf = malloc(strlen(str)+1);
  if (buf) {
    for (i=0; i<sizeof(str)-1; i++)
      buf[i] = str[i];
    buf[i] = '\0';
  }
  puts(buf);
  for (i=0; i<argc; i++)
    puts(argv[i]);
  return 0;
}
```

オフバイワンエラーと文字列

脆弱なコード：

```
#include <stdio.h>
#include <string.h>

int main(void)
{
#define LEN 16
  char a[LEN], b[LEN], c[LEN*2];

  strcpy(a, "0123456789abcdef");
  strcpy(b, "0123456789abcdef");
  strcpy(c, a);
  strcat(c, b);
  puts(c);
  return 0;
}
```

適切な strncpy, strncat 用法

脆弱性への対処：secure005.c

```
#include <stdio.h>
#include <string.h>

int main(void)
{
#define LEN 16
  char a[LEN], b[LEN], c[LEN*2];

  strncpy(a, "0123456789abcdef", sizeof(a)); a[sizeof(a)-1]='\0';
  strncpy(b, "0123456789abcdef", sizeof(b)); b[sizeof(b)-1]='\0';
  strncpy(c, a, sizeof(c)); c[sizeof(c)-1] = '\0';
  strncat(c, b, sizeof(c)-strlen(c)-1);
  puts(c);
  return 0;
}
```

簡単な strlcpy, strlcat 用法

脆弱性への対処：secure006.c

```
#include <stdio.h>
#include <string.h>

int main(void)
{
#define LEN 16
  char a[LEN], b[LEN], c[LEN*2];

  strlcpy(a, "0123456789abcdef", sizeof(a));
  strlcpy(b, "0123456789abcdef", sizeof(b));
  strlcpy(c, a, sizeof(c));
  strlcat(c, b, sizeof(c));
  puts(c);
  return 0;
}
```

不適切なメモリ操作：memcpy

脆弱なコード：

```
#include <stdio.h>
#include <string.h>

char buf[] = "0123456789abcdef";

int get_buf(char *str)
{
  memcpy(buf, str, strlen(str));
  return 0;
}

int main(int argc, char *argv[])
{
  if (argc == 1+1) {
    get_buf(argv[1]);
  }
  puts(buf);
  return 0;
}
```

適切なメモリ操作

脆弱性への対処：secure007.c

```
#include <stdio.h>
#include <string.h>

char buf[] = "0123456789abcdef";

int get_buf(char *str)
{
  size_t sz = strlen(str); /* memcpy ではコピー元と先のサイズに注意！ */
  memcpy(buf, str, (sizeof(buf)-1 <= sz ? sizeof(buf)-1 : sz));
  return 0;
}

int main(int argc, char *argv[])
{
  if (argc == 1+1) {
    get_buf(argv[1]);
  }
  puts(buf);
  return 0;
}
```

適切なメモリ操作

脆弱性への対処：secure007.c

```
#include <stdio.h>
#include <string.h>

/* 安全な memcpy_s を使おう！ しかし、無い場合は： */
#define memcpy_s(d, dz, s, sz) memcpy((d), (s), ((dz)<=(sz)?(dz):(sz)))

char buf[] = "0123456789abcdef";

int get_buf(char *str)
{
  size_t sz = strlen(str);
  memcpy_s(buf, sizeof(buf)-1, str, sz);
  return 0;
}

int main(int argc, char *argv[])
{
  if (argc == 1+1) {
    get_buf(argv[1]);
  }
  puts(buf);
  return 0;
}
```

不適切なメモリ操作：memmove

脆弱なコード：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

struct {
  char buf[16+1], foo[16+1];
} bar = {
  "0123456789abcdef", "****************",
};

int main(int argc, char *argv[])
{
  if (argc == 3+1) {
    int d = atoi(argv[1]), s = atoi(argv[2]), l = atoi(argv[3]);

    if (d < 0 || s < 0 || l < 0) return 1;
    memmove(bar.buf+d, bar.buf+s, l);
  }
  puts(bar.buf);
  return 0;
}
```

適切なメモリ操作

脆弱性への対処：secure008.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/* 安全な memmove_s を使おう！ しかし、無い場合は： */
#define memmove_s(d, dz, s, sz) memmove((d), (s), ((dz)<=(sz)?(dz):(sz)))

struct {
  char buf[16+1], foo[16+1];
} bar = {
  "0123456789abcdef", "****************",
};

int main(int argc, char *argv[])
{
  if (argc == 3+1) {
    int d = atoi(argv[1]), s = atoi(argv[2]), l = atoi(argv[3]);

    if (d < 0 || s < 0 || l < 0) return 1;
    memmove_s(bar.buf+d, sizeof(bar.buf)-d, bar.buf+s, l);
  }
  puts(bar.buf);
  return 0;
}
```

不適切なメモリ確保

脆弱なコード：

```
double **mat(int M, int N)
{
  int i;
  double **m;

  m = (double **)malloc(sizeof(double *)*M);
  for (i=0; i<M; i++)
    m[i] = (double *)malloc(sizeof(double)*N);
  return m;
}
  :
int main(int argc, char *argv[])
{
  int i, j, n = 3;
  double **a, **b, **c;

  if (argc == 1+1) {
    n = atoi(argv[1]);
    if (n < 0) return 1;
  }
  a = mat(n, n); b = mat(n, n); c = mat(n, n);
  :
}
```

適切なメモリ確保

脆弱性への対処：secure009.c

```
double **mat(int M, int N)
{
  int i;
  double **m;

  m = (double **)malloc(sizeof(double *)*M);
  if (!m)
    return NULL;
  for (i=0; i<M; i++) {
    m[i] = (double *)malloc(sizeof(double)*N);
    if (!m[i]) {
      do free(m[--i]); while (i != 0);
      free(m);
      return NULL;
    }
  }
  return m;
}
  :
  a = mat(n, n); b = mat(n, n); c = mat(n, n);
  /* メモリ確保の成否を検査すべき！ */
  if (!a || !b || !c) { fprintf(stderr, "malloc failed\n"); return 1; }
  :
```

不適切なメモリ操作

脆弱なコード：

```
void mat_mult_mat(int M, int N, int L, double **a, double **b, double **c)
{
  int i, j, k;

  for (i=0; i<M; i++)
    for (j=0; j<L; j++)
      for (k=0; k<N; k++)
        c[i][j] += a[i][k]*b[k][j];
}
int main(void)
{
  int i, j;
  double **a = mat(3, 3), **b = mat(3, 3), **c = mat(3, 3);

  unit_mat(3, 3, a);
  unit_mat(3, 3, b);
  mat_mult_mat(3, 3, 3, a, b, c);
  for (i=0; i<3; i++) {
    for (j=0; j<3; j++)
      printf("%g ", c[i][j]);
    printf("\n");
  }
  return 0;
}
```

適切なメモリ操作

脆弱性への対処：secure010.c

```
void mat_mult_mat(int M, int N, int L, double **a, double **b, double **c)
{
  int i, j, k;

  for (i=0; i<M; i++)
    for (j=0; j<L; j++)
      for (c[i][j] = 0, k=0; k<N; k++)
        c[i][j] += a[i][k]*b[k][j];
} /* 初期化されていないメモリ参照への対処 */
int main(void)
{
  int i, j;
  double **a = mat(3, 3), **b = mat(3, 3), **c = mat(3, 3);

  unit_mat(3, 3, a);
  unit_mat(3, 3, b);
  mat_mult_mat(3, 3, 3, a, b, c);
  for (i=0; i<3; i++) {
    for (j=0; j<3; j++)
      printf("%g ", c[i][j]);
    printf("\n");
  }
  return 0;
}
```

不適切なメモリ操作

脆弱なコード：

```
struct list {
  char buf[1024];
  struct list *next;
} *list_p = NULL; /* 順方向リスト（スタック） */

struct list *list_append(struct list item)
{
  struct list *me = malloc(sizeof(struct list));

  if (!me) return NULL;
  *me = item;
  me->next = list_p;
  return (list_p = me);
}
void list_free(void)
{
  struct list *p;

  p = list_p;
  while (p) {
    free(p);
    p = p->next;
  }
}
```

不適切なメモリ操作（つづき）

脆弱なコード（つづき）：

```
void list_print(void)
{
  struct list *p;

  p = list_p;
  while (p) {
    printf("%s", p->buf);
    p = p->next;
  }
}
int main(void)
{
  struct list item;

  while (fgets(item.buf, sizeof(item.buf), stdin)) {
    if (!list_append(item))
      return 1;
  }
  list_print();
  list_free();
  return 0;
}
```

適切なメモリ操作

脆弱性への対処：secure011.c

```
struct list {
  char buf[1024];
  struct list *next;
} *list_p = NULL; /* 順方向リスト（スタック） */

struct list *list_append(struct list item)
{
  struct list *me = malloc(sizeof(struct list));

  if (!me) return NULL;
  *me = item;
  me->next = list_p;
  return (list_p = me);
}
void list_free(void)
{
  struct list *p, *q;

  p = list_p;
  while (p) {
    q = p->next;
    free(p);
    p = q;
  } /* 解放したメモリ参照への対処 */
}
```

不適切なメモリ解放

脆弱なコード：

```
struct list {
  char *str;
  struct list *next;
} *list_p = NULL, *list_q = NULL; /* 順方向リスト */

struct list *list_append(struct list item)
{
  struct list *me = malloc(sizeof(struct list));

  if (!me) return NULL;
  *me = item;
  me->next = list_p;
  return (list_p = me);
}
struct list *list_prepend(struct list item)
  :
void list_free(void)
{
  struct list *p, *q;

  for (p=list_p; p; p=q) {
    q = p->next;
    free(p->str);
    free(p);
  }
}
```

不適切なメモリ解放（つづき）

脆弱なコード（つづき）：

```
void list_print(void)
  :
int main(void)
{
  char buf[1024];
  struct list item;

  while (fgets(buf, sizeof(buf), stdin)) {
    item.str = malloc(strlen(buf)+1); /* 必要な長さだけメモリ確保 */
    if (item.str)
      strcpy(item.str, buf);
    if (!list_append(item))
      return 1;
    if (!list_prepend(item))
      return 1;
  }
  list_print();
  list_free();
  return 0;
}
```

適切なメモリ解放

脆弱性への対処：secure012.c

```
struct list {
  char *str;
  struct list *next;
} *list_p = NULL, *list_q = NULL; /* 順方向リスト */

struct list *list_append(struct list item)
{
  struct list *me = malloc(sizeof(struct list));

  if (!me) return NULL;
  me->str = strdup(item.str); /* ポインタではなく内容をコピー */
  me->next = list_p;
  return (list_p = me);
}
struct list *list_prepend(struct list item)
  :
void list_free(void)
{
  struct list *p, *q;

  for (p=list_p; p; p=q) {
    q = p->next;
    free(p->str); /* ここでの脅威が起こらなくなる */
    free(p);
  }
} /* メモリの二重解放への対処 */
```

不適切な整数の用法

脆弱なコード：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
  unsigned short len;
  char *buf = NULL;

  if (argc == 2+1) {
    len = strlen(argv[1]) + strlen(argv[2]);
    buf = malloc(len + 1);
    if (!buf) return 1;
    strcpy(buf, argv[1]);
    strcat(buf, argv[2]);
  }
  puts(buf);
  return 0;
}
```

適切な整数の用法

脆弱性への対処：secure013.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
  size_t len; /* 適切な整数型へ */
  char *buf = NULL;

  if (argc == 2+1) {
    len = strlen(argv[1]) + strlen(argv[2]);
    buf = malloc(len + 1);
    if (!buf) return 1;
    strcpy(buf, argv[1]);
    strcat(buf, argv[2]);
  }
  puts(buf);
  return 0;
}
```

不適切な整数の用法

脆弱なコード：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUF_SIZE 8

int main(int argc, char *argv[])
{
  int len;
  char buf[BUF_SIZE] = "";

  if (argc == 2+1) {
    len = atoi(argv[1]);
    if (len < BUF_SIZE)
      memcpy(buf, argv[2], len);
  }
  puts(buf);
  return 0;
}
```

適切な整数の用法

脆弱性への対処：secure014.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUF_SIZE 8

int main(int argc, char *argv[])
{
  int len;
  char buf[BUF_SIZE] = "";

  if (argc == 2+1) {
    len = atoi(argv[1]);
    if (0 < len && len < BUF_SIZE) /* 負値についても検査 */
      memcpy(buf, argv[2], len);
  }
  puts(buf);
  return 0;
}
```

不適切な書式付き出力の用法

脆弱なコード：

```
#include <stdio.h>
#include <stdlib.h>

void error_exit(char *me, char *msg)
{
  fprintf(stderr, "%s: %s\n", me, msg);
  exit(1);
}

int main(int argc, char *argv[])
{
  char msg[64];

  if (argc == 1+1) {
    sprintf(msg, "Illegal option(%s)", argv[1]);
    error_exit(argv[0], msg);
  }
  return 0;
}
```

適切な書式付き出力の用法

脆弱性への対処：secure015.c

```
#include <stdio.h>
#include <stdlib.h>

void error_exit(char *me, char *msg)
{
  fprintf(stderr, "%s: %s\n", me, msg);
  exit(1);
}

int main(int argc, char *argv[])
{
  char msg[64];

  if (argc == 1+1) { /* snprintf を使おう！ */
    snprintf(msg, sizeof(msg), "Illegal option(%s)", argv[1]);
    error_exit(argv[0], msg);
  }
  return 0;
}
```

適切な書式付き出力の用法

脆弱性への対処：secure015.c

```
#include <stdio.h>
#include <stdlib.h>

void error_exit(char *me, char *msg)
{
  fprintf(stderr, "%s: %s\n", me, msg);
  exit(1);
}

int main(int argc, char *argv[])
{
  char msg[64];

  if (argc == 1+1) { /* snprintf が無い場合は、面倒… */
    sprintf(msg, "Illegal option(%.*s)", (int)sizeof(msg)-(16+1), argv[1]);
    error_exit(argv[0], msg);
  }
  return 0;
} /* もしくは msg を動的にメモリ確保すべき */
```

不適切な書式付き出力の用法

脆弱なコード：

```
#include <stdio.h>
#include <stdlib.h>

void error_exit(char *me, int error_no, char *info)
{
  char *fmt[] = {
    "%s: file %s not found\n",
    "%s: file %s not readable\n",
  }, buf[1024];

  snprintf(buf, sizeof(buf), fmt[error_no], me, info);
  fprintf(stderr, buf);
  exit(1);
}

int main(int argc, char *argv[])
{
  if (argc == 1+1) {
    error_exit(argv[0], 1, argv[1]);
  }
  return 0;
}
```

適切な書式付き出力の用法

脆弱性への対処：secure015.c

```
#include <stdio.h>
#include <stdlib.h>

void error_exit(char *me, int error_no, char *info)
{
  char *fmt[] = {
    "%s: file %s not found\n",
    "%s: file %s not readable\n",
  }, buf[1024];

  snprintf(buf, sizeof(buf), fmt[error_no], me, info);
  fprintf(stderr, "%s", buf);
  exit(1); /* 書式にユーザからの入力がそのまま混入しないように対処 */
}

int main(int argc, char *argv[])
{
  if (argc == 1+1) {
    error_exit(argv[0], 1, argv[1]);
  }
  return 0;
}
```

不適切な書式付き入力の用法

脆弱なコード：

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  char name[8+1] = "";

  if (argc == 1+1) {
    sscanf(argv[1], "%s", name);
  }
  puts(name);
  return 0;
}
```

適切な書式付き入力の用法

脆弱性への対処：secure015.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  char name[8+1] = "";

  if (argc == 1+1) {
    sscanf(argv[1], "%8s", name); /* 無制限コピーへの対処 */
  }
  puts(name);
  return 0;
} /* しかし、書式付き入力にて文字列への代入は避けた方が無難 */
```

Cにおけるセキュアプログラミング：まとめ

Cプログラムに対する脅威から身を守るには：

- レガシーな非推奨関数は使わない
- Ｃ９９準拠の新しめの関数を使う
- 間違い難い安全な関数を使う・用意する
- メモリ管理を適切に行なう
  （スタックとヒープ、配列の添字、ポインタ、整数型、メモリ解放）
- Cにおける文字列（ヌル終端）と周辺の関数を理解する (test000〜013.c)
- Cにおける整数型（符合とサイズ）の特性を理解する (test014.c)
- 書式付き入出力関数は適切に使う
- 加えて、並列性に係る競合状態における脅威へ対処（スレッドセーフ、TOCTOU など）
- 脅威の緩和方法（最小権限での実行、入力データの無害化など）を検討する
- 言語仕様（暗黙の型変換など）、ＯＳ仕様（ファイルシステムなど）、サービス形態（ライブラリ、ウェブなど）に併せた対策が必要
- 詳しくは、「Ｃ／Ｃ＋＋セキュアコーディング（歌代 監訳、アスキー）」他