Name : Mohammed Tousif
Roll : CB.EN.P2CYS22008

SEI CERT C CODING STANDARD

RULE : ENVIRONMENT (ENV)

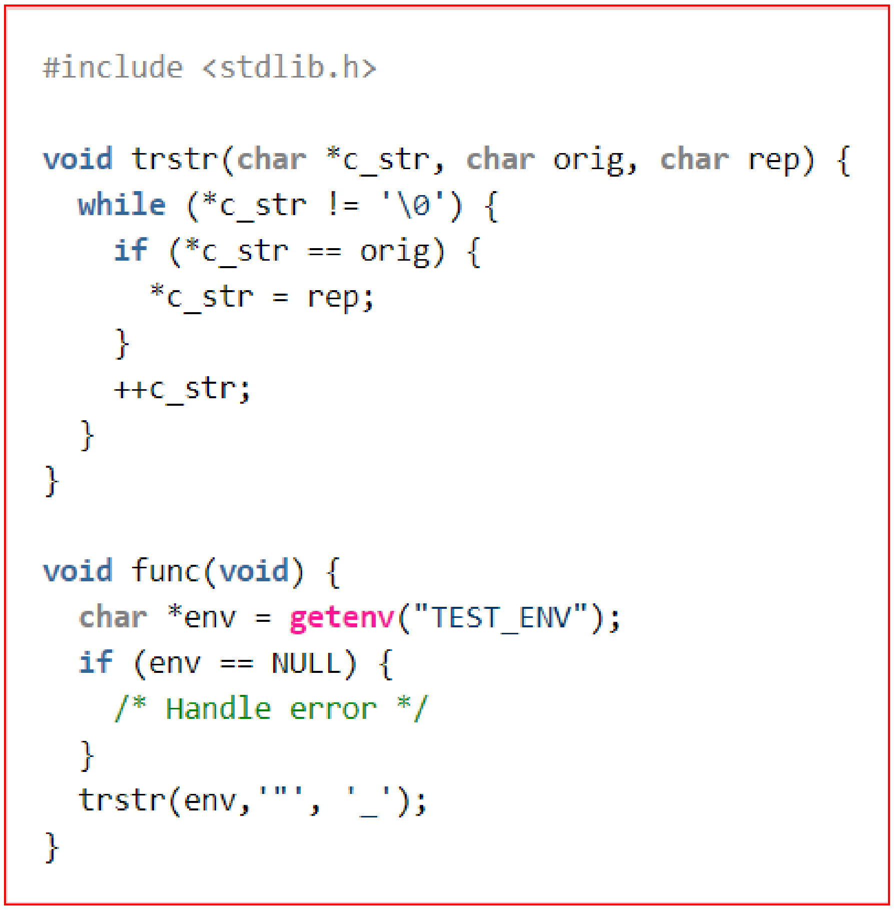
Contents :

ENV30-C. Do not modify the object referenced by the return value of certain functions.

Some functions such as getenv() and localeconv() will return a pointer to an object that cannot be modified without causing undefined behaviour.

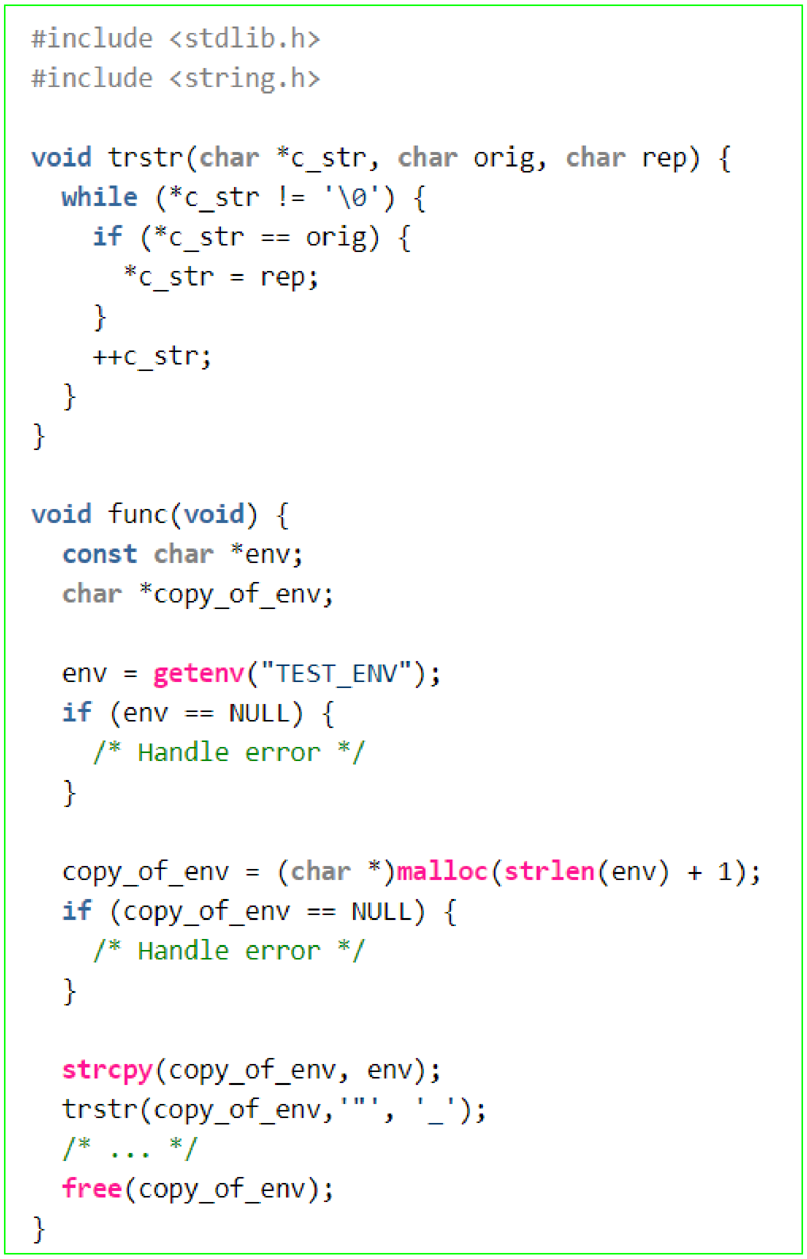
Noncomplaint Code Example (getenv()) :

The string returned by getenv() will be modified by replacing all double quotations(” “) into underscores (_).



```c
#include <stdlib.h>

void trstr(char *c_str, char orig, char rep) {
  while (*c_str != '\0') {
    if (*c_str == orig) {
      *c_str = rep;
    }
    ++c_str;
  }
}

void func(void) {
  char *env = getenv("TEST_ENV");
  if (env == NULL) {
    /* Handle error */
  }
  trstr(env,'"', '_');
}
```

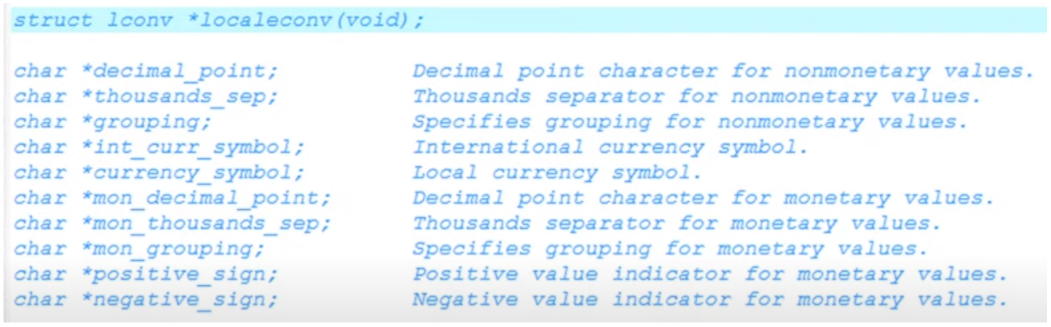
Complaint Solution (getenv()) :



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

void trstr(char *c_str, char orig, char rep) {
  while (*c_str != '\0') {
    if (*c_str == orig) {
      *c_str = rep;
    }
    ++c_str;
  }
}

void func(void) {
  const char *env;
  char *copy_of_env;

  env = getenv("TEST_ENV");
  if (env == NULL) {
    /* Handle error */
  }

  copy_of_env = (char *)malloc(strlen(env) + 1);
  if (copy_of_env == NULL) {
    /* Handle error */
  }

  strcpy(copy_of_env, env);
  trstr(copy_of_env,'"', '_');
  /* ... */
  free(copy_of_env);
}
```

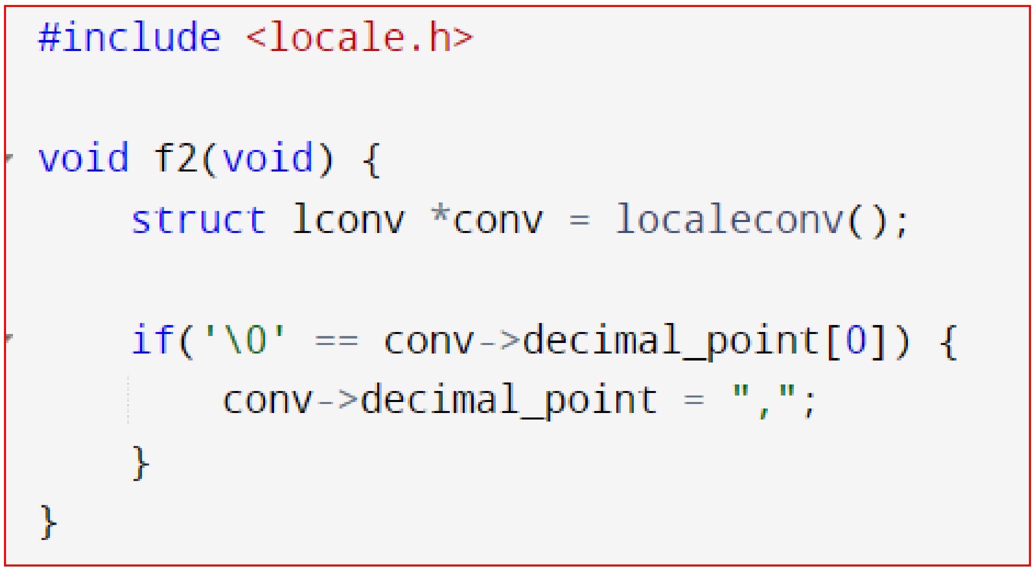
localeconv() :

The localeconv function obtains a pointer to a static object of type lconv, which represents numeric and monetary formatting rules of the current C locale.



```
struct lconv *localeconv(void);

char *decimal_point;          Decimal point character for nonmonetary values.
char *thousands_sep;          Thousands separator for nonmonetary values.
char *grouping;               Specifies grouping for nonmonetary values.
char *int_curr_symbol;        International currency symbol.
char *currency_symbol;        Local currency symbol.
char *mon_decimal_point;      Decimal point character for monetary values.
char *mon_thousands_sep;      Thousands separator for monetary values.
char *mon_grouping;           Specifies grouping for monetary values.
char *positive_sign;          Positive value indicator for monetary values.
char *negative_sign;          Negative value indicator for monetary values.
```

Noncomplaint Code Example (localeconv()) :



```c
#include <locale.h>

void f2(void) {
    struct lconv *conv = localeconv();

    if('\0' == conv->decimal_point[0]) {
        conv->decimal_point = ",";
    }
}
```

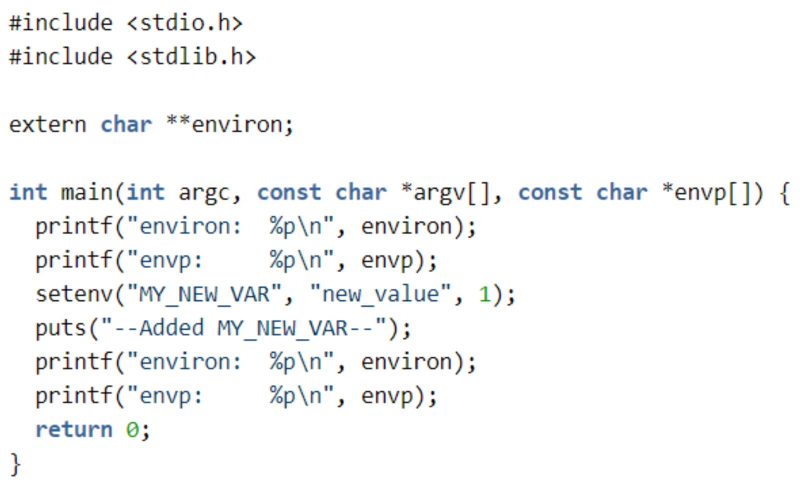
Complaint Solution (localeconv()) :

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

void f2(void) {
  const struct lconv *conv = localeconv();
  if (conv == NULL) {
     /* Handle error */
  }

  struct lconv *copy_of_conv = (struct lconv *)malloc(
    sizeof(struct lconv));
  if (copy_of_conv == NULL) {
    /* Handle error */
  }

  memcpy(copy_of_conv, conv, sizeof(struct lconv));

  if ('\0' == copy_of_conv->decimal_point[0]) {
    copy_of_conv->decimal_point = ".";
  }
  /* ... */
  free(copy_of_conv);
}
```

ENV31-C. Do not rely on an environment pointer following an operation that may invalidate it.

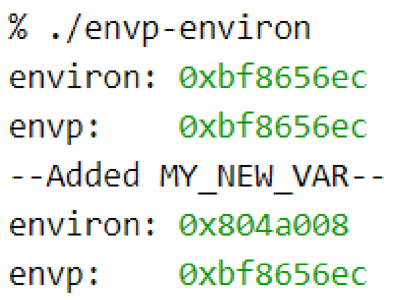
Under a hosted environment , it is possible to access the environment through a modified form of main()

However , modifying the environment by any means may cause the environment to be relocated , with the result that envp now references an incorrect location. for ex ,



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

extern char **environ;

int main(int argc, const char *argv[], const char *envp[]) {
  printf("environ:  %p\n", environ);
  printf("envp:     %p\n", envp);
  setenv("MY_NEW_VAR", "new_value", 1);
  puts("--Added MY_NEW_VAR--");
  printf("environ:  %p\n", environ);
  printf("envp:     %p\n", envp);
  return 0;
}
```

it yields ,



```
% ./envp-environ
environ: 0xbf8656ec
envp:    0xbf8656ec
--Added MY_NEW_VAR--
environ: 0x804a008
envp:    0xbf8656ec
```

Noncomplaint Code Example :

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

int main(int argc, const char *argv[], const char *envp[]) {
  if (setenv("MY_NEW_VAR", "new_value", 1) != 0) {
    /* Handle error */
  }
  if (envp != NULL) {
    for (size_t i = 0; envp[i] != NULL; ++i) {
      puts(envp[i]);
    }
  }
  return 0;
}
```

Accessing the **envp** pointer after calling setenv(). It may no longer point to the current environment. Thus , having unanticipated behaviour.

Complaint Solution :

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

extern char **environ;

int main(void) {
  if (setenv("MY_NEW_VAR", "new_value", 1) != 0) {
    /* Handle error */
  }
  if (environ != NULL) {
    for (size_t i = 0; environ[i] != NULL; ++i) {
      puts(environ[i]);
    }
  }
  return 0;
}
```

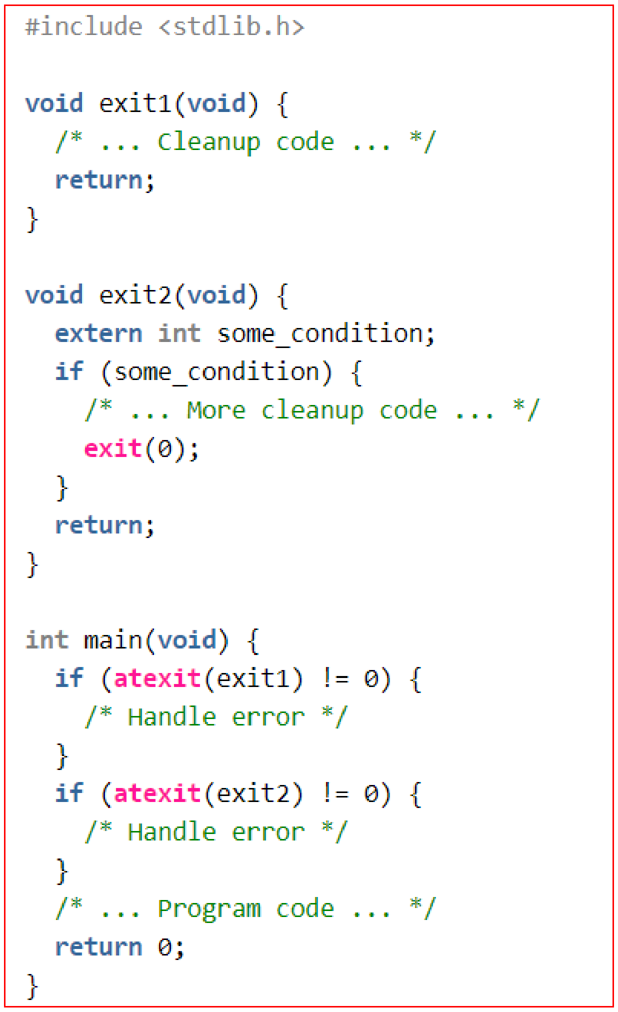
ENV32-C. All exit handlers must return normally

C Standard provides functions that cause an application to terminate normally such as exit() , quick_exit() and _Exit(). These are called **exit functions**.

When exit() is called, the control transfers out of main() and calls atexit(). which called as **exit handlers**.

A nested call to an exit function is undefined behaviour.

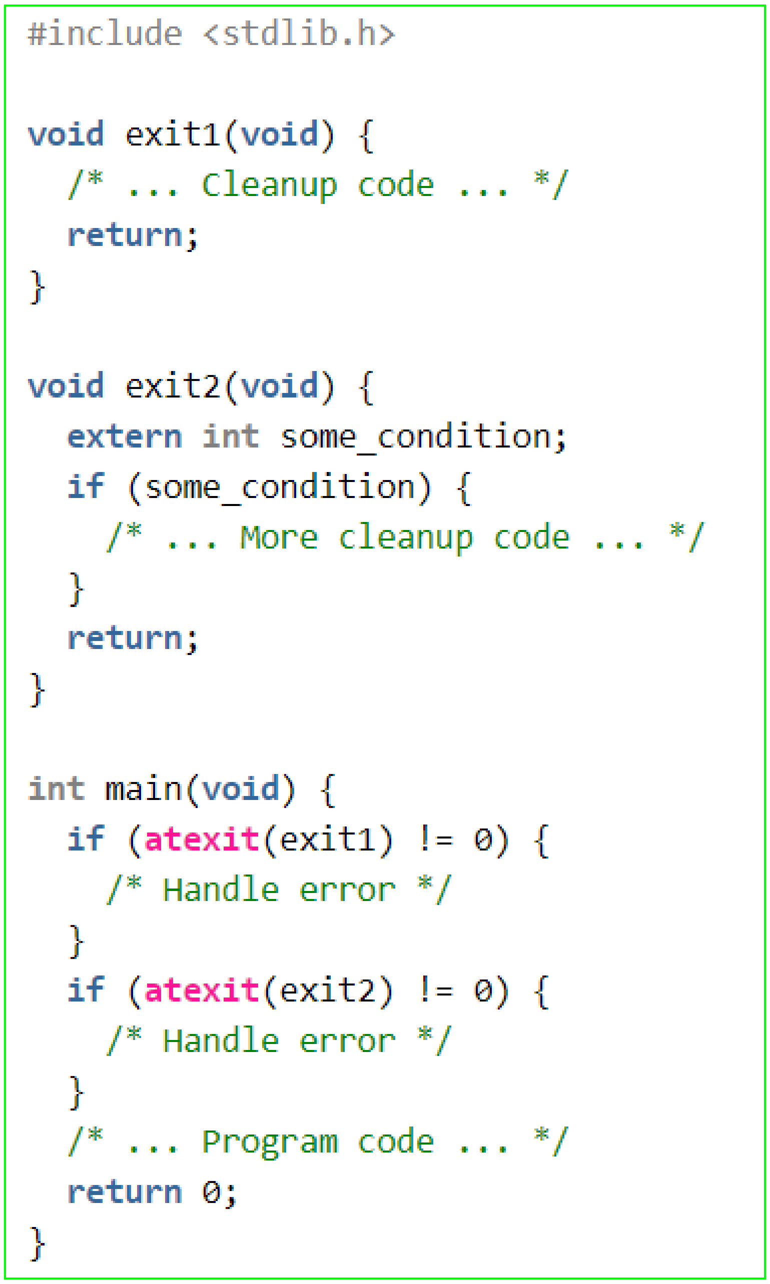
Noncomplaint Code Example :



```c
#include <stdlib.h>

void exit1(void) {
  /* ... Cleanup code ... */
  return;
}

void exit2(void) {
  extern int some_condition;
  if (some_condition) {
    /* ... More cleanup code ... */
    exit(0);
  }
  return;
}

int main(void) {
  if (atexit(exit1) != 0) {
    /* Handle error */
  }
  if (atexit(exit2) != 0) {
    /* Handle error */
  }
  /* ... Program code ... */
  return 0;
}
```

Complaint Solution :

exit handler atexit() must exit by returning.



```
#include <stdlib.h>

void exit1(void) {
  /* ... Cleanup code ... */
  return;
}

void exit2(void) {
  extern int some_condition;
  if (some_condition) {
    /* ... More cleanup code ... */
  }
  return;
}

int main(void) {
  if (atexit(exit1) != 0) {
    /* Handle error */
  }
  if (atexit(exit2) != 0) {
    /* Handle error */
  }
  /* ... Program code ... */
  return 0;
}
```

ENV33-C. Do not call system()

The C standard system() function executes a specific command by invoking an command processor such as UNIX shell.

Use of system() function can result in exploitable vulnerabilities such as taking control over current working directory , getting root shell , shellshock vulnerability , executing commands etc.

Noncomplaint Code Example :

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

enum { BUFFERSIZE = 512 };

void func(const char *input) {
  char cmdbuf[BUFFERSIZE];
  int len_wanted = snprintf(cmdbuf, BUFFERSIZE,
                            "any_cmd '%s'", input);
  if (len_wanted >= BUFFERSIZE) {
    /* Handle error */
  } else if (len_wanted < 0) {
    /* Handle error */
  } else if (system(cmdbuf) == -1) {
    /* Handle error */
  }
}
```

If this code compiles, attacker can give command like this :

```
happy'; useradd 'attacker
```

Shell would interpret this string as two seperate commands :

```
any_cmd 'happy';
useradd 'attacker'
```

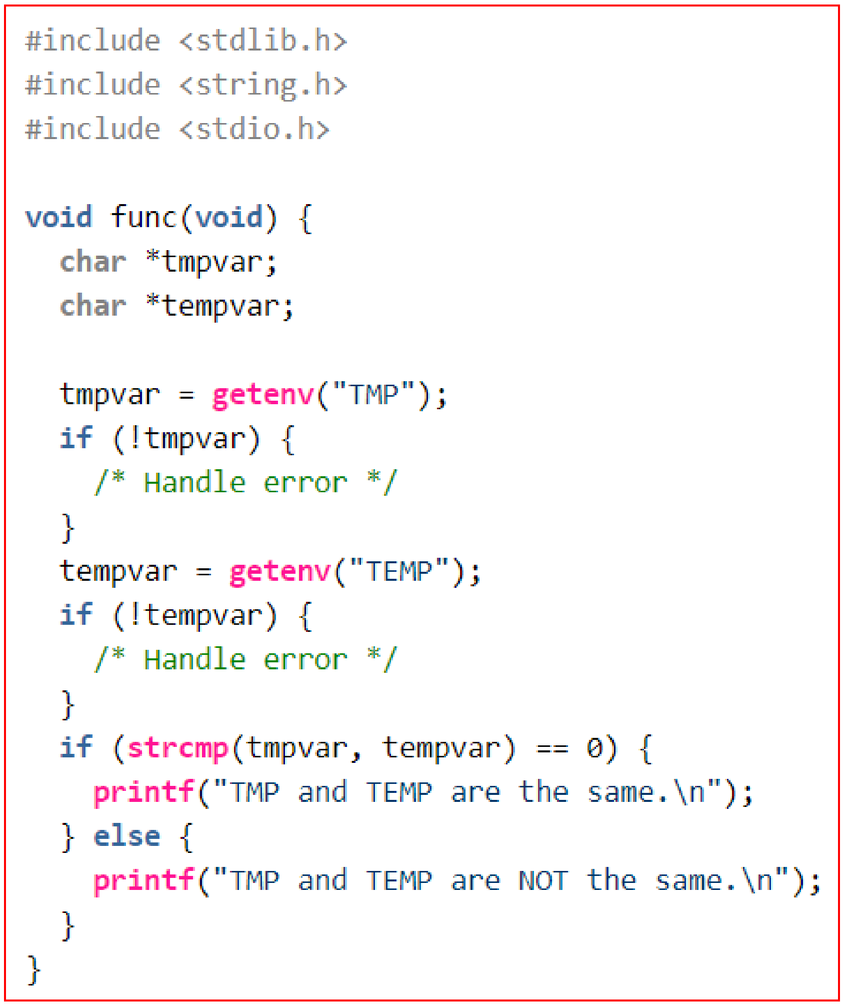
Complaint Solution : Using execve().

```c
#include <errno.h>
#include <stdlib.h>

void func(char *input) {
  pid_t pid;
  int status;
  pid_t ret;
  char *const args[3] = {"any_exe", input, NULL};
  char **env;
  extern char **environ;

  /* ... Sanitize arguments ... */

  pid = fork();
  if (pid == -1) {
    /* Handle error */
  } else if (pid != 0) {
    while ((ret = waitpid(pid, &status, 0)) == -1) {
      if (errno != EINTR) {
        /* Handle error */
        break;
      }
    }
    if ((ret == 0) ||
        !(WIFEXITED(status) && !WEXITSTATUS(status))) {
      /* Report unexpected child status */
    }
  } else {
    /* ... Initialize env as a sanitized copy of environ ... */
    if (execve("/usr/bin/any_cmd", args, env) == -1) {
      /* Handle error */
      _Exit(127);
    }
  }
```

ENV34-C. Do not store pointers returned by certain functions.

Function such as getenv returns a pointer to a statically allocated buffer. We should not store this pointer because the string data it points to may be overwritten by another call of getnev() which causes undefined behaviour.

Noncomplaint Code Example :



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

void func(void) {
  char *tmpvar;
  char *tempvar;

  tmpvar = getenv("TMP");
  if (!tmpvar) {
    /* Handle error */
  }
  tempvar = getenv("TEMP");
  if (!tempvar) {
    /* Handle error */
  }
  if (strcmp(tmpvar, tempvar) == 0) {
    printf("TMP and TEMP are the same.\n");
  } else {
    printf("TMP and TEMP are NOT the same.\n");
  }
}
```

Here , it shows both variables are same even though they are different values.

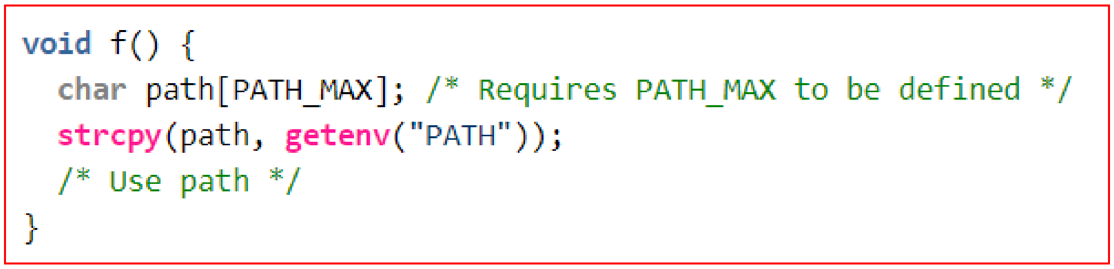
Complaint Solution :

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

void func(void) {
  char *tmpvar;
  char *tempvar;

  const char *temp = getenv("TMP");
  if (temp != NULL) {
    tmpvar = (char *)malloc(strlen(temp)+1);
    if (tmpvar != NULL) {
      strcpy(tmpvar, temp);
    } else {
      /* Handle error */
    }
  } else {
    /* Handle error */
  }

  temp = getenv("TEMP");
  if (temp != NULL) {
    tempvar = (char *)malloc(strlen(temp)+1);
    if (tempvar != NULL) {
      strcpy(tempvar, temp);
    } else {
      /* Handle error */
    }
  } else {
    /* Handle error */
  }

  if (strcmp(tmpvar, tempvar) == 0) {
    printf("TMP and TEMP are the same.\n");
  } else {
    printf("TMP and TEMP are NOT the same.\n");
  }
  free(tmpvar);
  free(tempvar);
```

ENV35-C. Do not make assumptions about the size of an environment variable.

If the environment variable needs to be stored, the length of associated string should be calculated and the storage dynamically allocated.

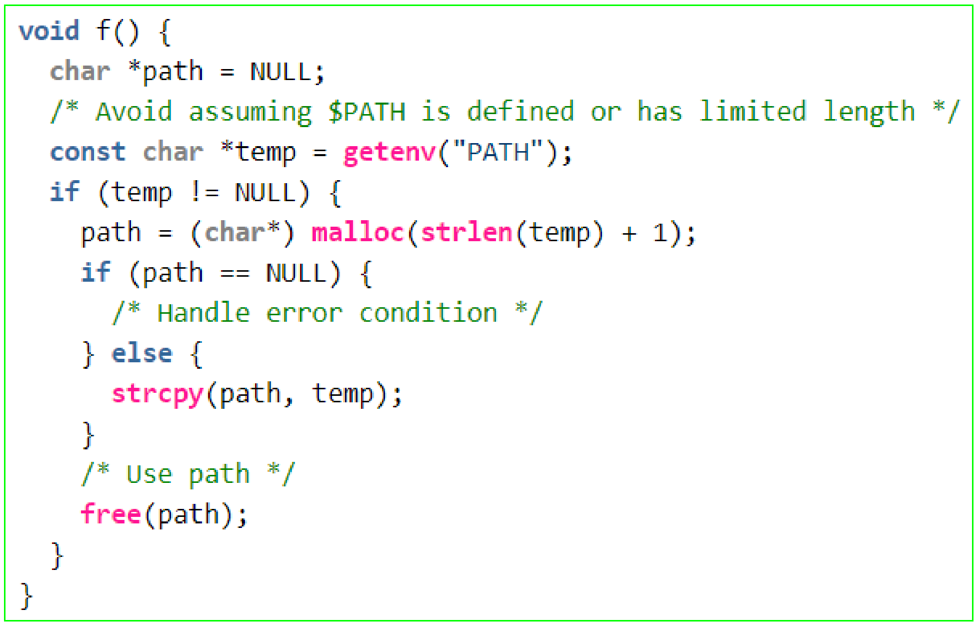
Noncomplaint Code Example :



```c
void f() {
  char path[PATH_MAX]; /* Requires PATH_MAX to be defined */
  strcpy(path, getenv("PATH"));
  /* Use path */
}
```

Complaint Solution :

Calculating size of the string using strlen().

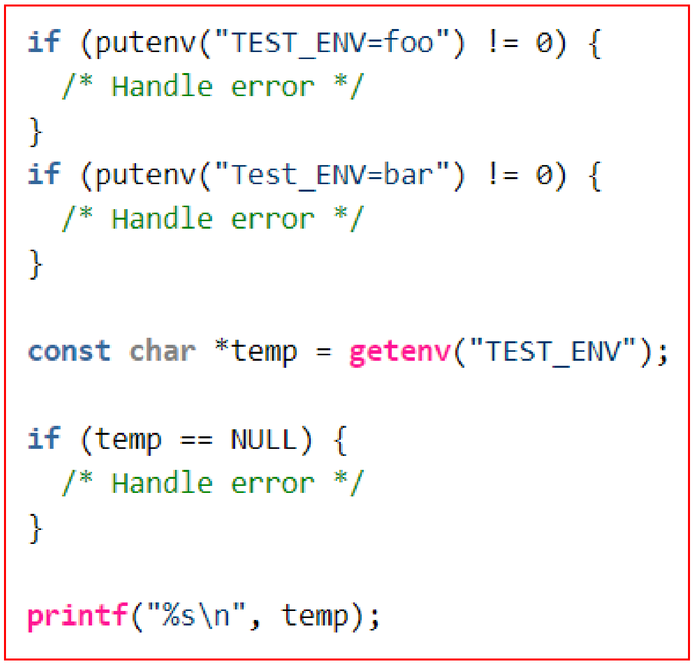


```c
void f() {
  char *path = NULL;
  /* Avoid assuming $PATH is defined or has limited length */
  const char *temp = getenv("PATH");
  if (temp != NULL) {
    path = (char*) malloc(strlen(temp) + 1);
    if (path == NULL) {
      /* Handle error condition */
    } else {
      strcpy(path, temp);
    }
    /* Use path */
    free(path);
  }
}
```

ENV36-C. Beware of multiple environment variables with same effective name

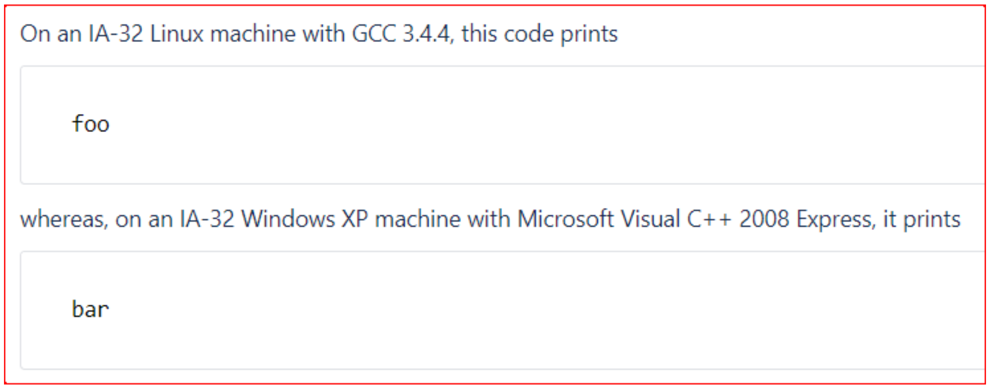
Depending on the implementation, multiple environment variables woth same name may be allowed and can cause unexpected results like different implementations due to case sensitive in UNIX and Windows XP.

Noncomplaint Code Example :



```
if (putenv("TEST_ENV=foo") != 0) {
  /* Handle error */
}
if (putenv("Test_ENV=bar") != 0) {
  /* Handle error */
}

const char *temp = getenv("TEST_ENV");

if (temp == NULL) {
  /* Handle error */
}

printf("%s\n", temp);
```

This behaves differently when compiled and run on Linux and Windows platforms.

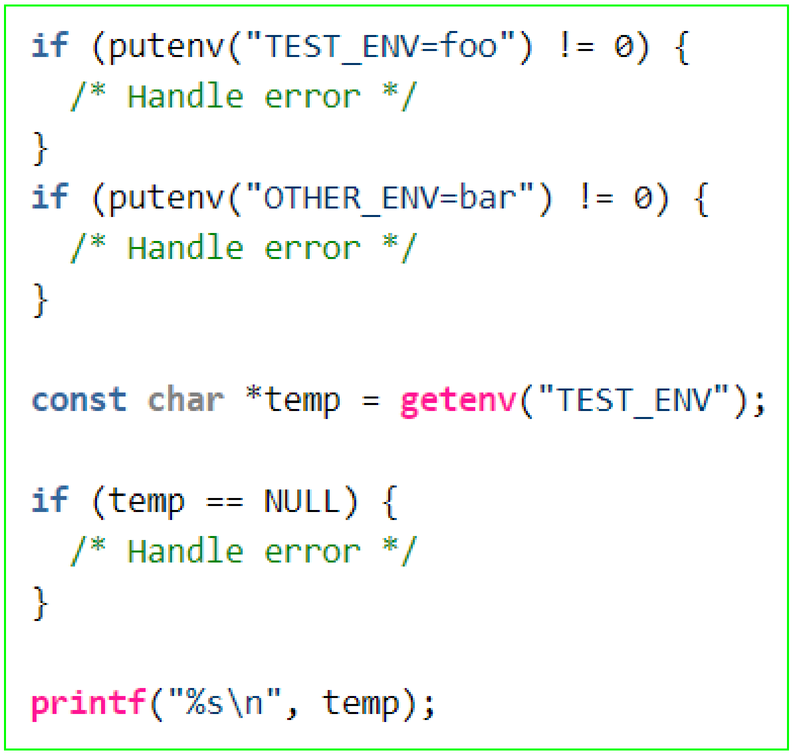

On an IA-32 Linux machine with GCC 3.4.4, this code prints

```
foo
```

whereas, on an IA-32 Windows XP machine with Microsoft Visual C++ 2008 Express, it prints

```
bar
```

Complaint Solution :



```c
if (putenv("TEST_ENV=foo") != 0) {
  /* Handle error */
}
if (putenv("OTHER_ENV=bar") != 0) {
  /* Handle error */
}

const char *temp = getenv("TEST_ENV");

if (temp == NULL) {
  /* Handle error */
}

printf("%s\n", temp);
```

ENV37-C. Sanitize the environment when invoking external programs

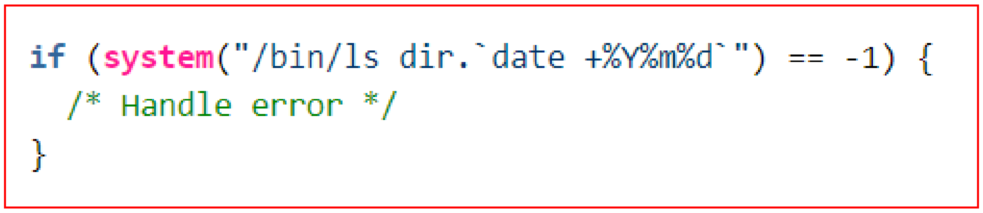
Many programs and libraries depend on environment variable settings. Because env variables inherited from parent process and attacker can sabotage variables causing undefined behaviour.

Programs with higher privileges like setuid can cause undefined behaviour.

The best practice for such programs is to :

- Drop privileges once they are no longer necessary.
- Avoid calling system().
- Clear the environment and fill it with default values.

Noncomplaint Code Example :

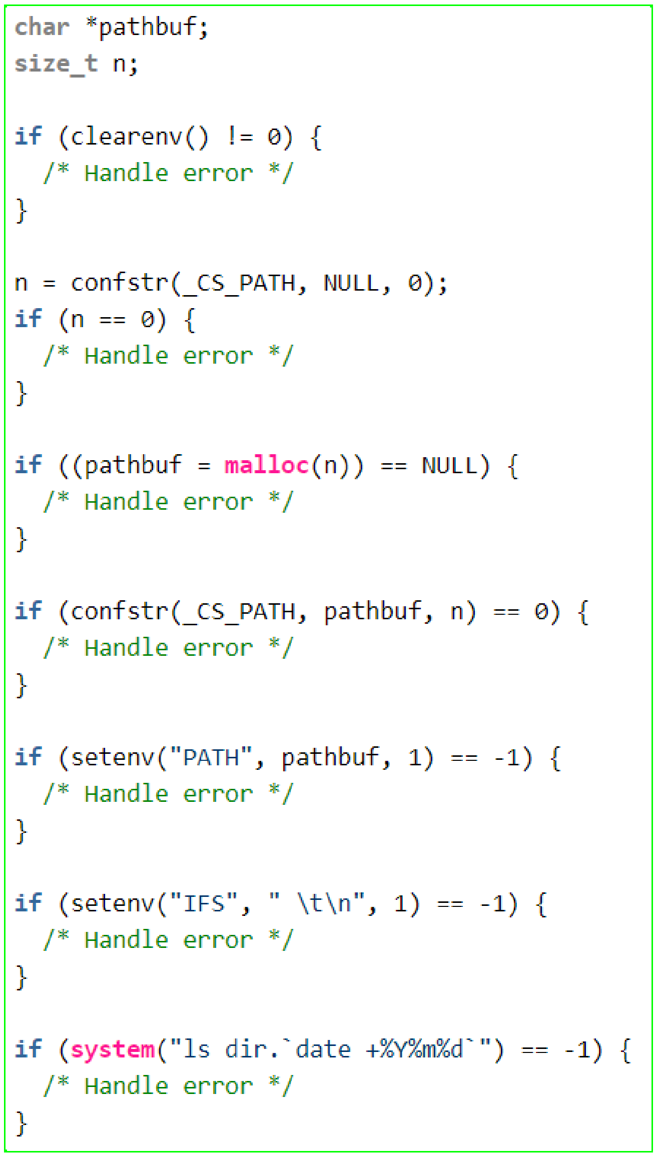


```
if (system("/bin/ls dir.`date +%Y%m%d`") == -1) {
  /* Handle error */
}
```

It invokes system() executing the /bin/ls. Although IFS does not affect the command portion , it does determine how argument built after calling date.

An attacker can set IFS value to ”.” . Thus , the intended directory will not be found.

Compliant Solution :



```c
char *pathbuf;
size_t n;

if (clearenv() != 0) {
  /* Handle error */
}

n = confstr(_CS_PATH, NULL, 0);
if (n == 0) {
  /* Handle error */
}

if ((pathbuf = malloc(n)) == NULL) {
  /* Handle error */
}

if (confstr(_CS_PATH, pathbuf, n) == 0) {
  /* Handle error */
}

if (setenv("PATH", pathbuf, 1) == -1) {
  /* Handle error */
}

if (setenv("IFS", " \t\n", 1) == -1) {
  /* Handle error */
}

if (system("ls dir.`date +%Y%m%d`") == -1) {
  /* Handle error */
}
```

clearenv() is used to clear out the environment where available and env variables such as **PATH** and **IFS** are set to safe values before system() is invoked. Thus sanitizing the environment and its variables.