String kfuncs

Simplifying string handling in eBPF programs

Viktor Malík
Principal Software Engineer @ Red Hat

January 31, 2026

Red Hat

Motivation: Why string handling?

- Many hooks (functions, LSMs, skb data) allow to access strings from eBPF programs
- Examples where string handling is useful:
  - Matching filesystem paths (e.g., is the file inside `/home/$USER/...`?)
  - Matching process names (e.g., does `comm` contain `systemd`?)
  - Searching environment variables
  - Parsing protocol (e.g. HTTP) headers

Motivation: String processing, the old way

- Copy string from kernel memory using `bpf_probe_read_str()`
- Manually implement all string operations

Motivation: String processing, the old way

- Copy string from kernel memory using `bpf_probe_read_str()`
- Manually implement all string operations
- Example:

```
SEC("lsm/bprm_check_security")
int BPF_PROG(path_check, struct linux_binprm *bprm) {
    char path[32];
    int path_len, last_slash = -1;

    path_len = bpf_probe_read_str(path, sizeof(path), bprm->filename);
    if (path_len < 0  path_len >= 32) return 0;

    for (int i = 0; i < path_len; i++) {
        if (path[i] == '/')
            last_slash = i;
    }
    [...]
```

Motivation: String processing, the old way

- Copy string from kernel memory using `bpf_probe_read_str()`
- Manually implement all string operations
- Example:

```
SEC("lsm/bprm_check_security")
int BPF_PROG(path_check, struct linux_binprm *bprm) {
    char path[32];
    int path_len, last_slash = -1;

    path_len = bpf_probe_read_str(path, sizeof(path), bprm->filename);
    if (path_len < 0  path_len >= 32) return 0;

    for (int i = 0; i < path_len; i++) {
        if (path[i] == '/')
            last_slash = i;
    }
    [...]
```

Motivation: String processing, the old way

- Copy string from kernel memory using `bpf_probe_read_str()`
- Manually implement all string operations
- Example:

```
SEC("lsm/bprm_check_security")
int BPF_PROG(path_check, struct linux_binprm *bprm) {
    char path[32];
    int path_len, last_slash = -1;

    path_len = bpf_probe_read_str(path, sizeof(path), bprm->filename);
    if (path_len < 0  path_len >= 32) return 0;

    for (int i = 0; i < path_len; i++) {
        if (path[i] == '/')
            last_slash = i;
    }
    [...]
```

String kfuncs

- BPF Kernel Functions (kfuncs) are the modern way of exposing kernel functionality to eBPF programs
- **String kfuncs** provide implementations of the most common string operations

String kfuncs

- BPF Kernel Functions (kfuncs) are the modern way of exposing kernel functionality to eBPF programs
- **String kfuncs** provide implementations of the most common string operations
- Main advantages:
  - No need to reimplement the functions manually
  - No need to copy the strings onto the stack (which is limited) or to maps (which bring overhead)
  - Small performance benefit

Naive solution

- First idea: just call in-kernel implementations:

```
__bpf_kfunc int bpf_strlen(const char *s)
{
  return strlen(s);
}
```

```
size_t strlen(const char *s)
{
  const char *sc;
  for (sc = s; *sc != '\0'; ++sc)
    /* nothing */;
  return sc - s;
}
```

Naive solution

- First idea: just call in-kernel implementations:

```
__bpf_kfunc int bpf_strlen(const char *s)
{
  return strlen(s);
}
```

```
size_t strlen(const char *s)
{
  const char *sc;
  for (sc = s; *sc != '\0'; ++sc)
    /* nothing */;
  return sc - s;
}
```

- Not possible since we need to ensure:
  - **Safety**
    - Verifier is only able to check the first byte of the strings
    - Cannot use naked dereference as the memory doesn't have to be paged in

Naive solution

- First idea: just call in-kernel implementations:

```
__bpf_kfunc int bpf_strlen(const char *s)
{
  return strlen(s);
}
```

```
size_t strlen(const char *s)
{
  const char *sc;
  for (sc = s; *sc != '\0'; ++sc)
    /* nothing */;
  return sc - s;
}
```

- Not possible since we need to ensure:
  - **Safety**
    - Verifier is only able to check the first byte of the strings
    - Cannot use naked dereference as the memory doesn't have to be paged in
  - **Termination**
    - Strings are not necessarily null-terminated so the common algorithms can loop forever

Better solution

- Open code the functions:

```
__bpf_kfunc int bpf_strlen(const char *s__ign)
{
  guard(pagefault)();
  for (i = 0; i < XATTR_SIZE_MAX; i++) {
    __get_kernel_nofault(&c, s__ign, char, err_out);
    if (c == '\0') { return i; }
    s__ign++;
  }
  return i == XATTR_SIZE_MAX ? -E2BIG : i;
err_out:
  return -EFAULT;
}
```

Better solution

- Open code the functions:

```
__bpf_kfunc int bpf_strlen(const char *s__ign)
{
  guard(pagefault)();
  for (i = 0; i < XATTR_SIZE_MAX; i++) {
    __get_kernel_nofault(&c, s__ign, char, err_out);
    if (c == '\0') { return i; }
    s__ign++;
  }
  return i == XATTR_SIZE_MAX ? -E2BIG : i;
err_out:
  return -EFAULT;
}
```

- Features:
  - page faults disabled

Better solution

- Open code the functions:

```
__bpf_kfunc int bpf_strlen(const char *s__ign)
{
  guard(pagefault)();
  for (i = 0; i < XATTR_SIZE_MAX; i++) {
    __get_kernel_nofault(&c, s__ign, char, err_out);
    if (c == '\0') { return i; }
    s__ign++;
  }
  return i == XATTR_SIZE_MAX ? -E2BIG : i;
err_out:
  return -EFAULT;
}
```

- Features:
  - page faults disabled
  - `__get_kernel_nofault` instead of plain dereference

Better solution

- Open code the functions:

```
__bpf_kfunc int bpf_strlen(const char *s__ign)
{
  guard(pagefault)();
  for (i = 0; i < XATTR_SIZE_MAX; i++) {
    __get_kernel_nofault(&c, s__ign, char, err_out);
    if (c == '\0') { return i; }
    s__ign++;
  }
  return i == XATTR_SIZE_MAX ? -E2BIG : i;
err_out:
  return -EFAULT;
}
```

- Features:
  - page faults disabled
  - `__get_kernel_nofault` instead of plain dereference
  - upper limit on the string length

Verifier support

- String kfunc arguments:
  - Null-terminated (not necessarily) strings located in eBPF or kernel memory → unsafe pointers
  - Size arguments for bounded functions → general integers

Verifier support

- String kfunc arguments:
  - Null-terminated (not necessarily) strings located in eBPF or kernel memory → unsafe pointers
  - Size arguments for bounded functions → general integers
- Verifier supports "special suffices" for kfunc arguments
  - `___str` for string literals

Verifier support

- String kfunc arguments:
  - Null-terminated (not necessarily) strings located in eBPF or kernel memory → unsafe pointers
  - Size arguments for bounded functions → general integers
- Verifier supports "special suffices" for kfunc arguments
  - ~~`__str` for string literals~~
  - `__sz` for buffer size literals

Verifier support

- String kfunc arguments:
  - Null-terminated (not necessarily) strings located in eBPF or kernel memory → unsafe pointers
  - Size arguments for bounded functions → general integers
- Verifier supports "special suffices" for kfunc arguments
  - ~~`__str` for string literals~~
  - ~~`__sz` for buffer size literals~~
  - `__ign` for ignoring argument verification

Verifier support

- String kfunc arguments:
  - Null-terminated (not necessarily) strings located in eBPF or kernel memory → unsafe pointers
  - Size arguments for bounded functions → general integers
- Verifier supports "special suffices" for kfunc arguments
  - ~~`__str` for string literals~~
  - ~~`__sz` for buffer size literals~~
  - `__ign` for ignoring argument verification
- We can afford ignoring verification since safety is ensured dynamically

API details

- Read-only functions for now
- Initially added functions:
  - `bpf_strcmp`
  - `bpf_strlen`, `bpf_strnlen`
  - `bpf_strstr`, `bpf_strnstr`
  - `bpf_strchr`, `bpf_strnchr`, `bpf_strchrnul`, `bpf_strrchr`
  - `bpf_strspn`, `bpf_strcspn`
- Several new functions added after the initial patchset:
  - `bpf_strcasecmp`, `bpf_strncasecmp`
  - `bpf_strcasestr`, `bpf_strncasestr`

API details

Comparison with stdlib functions

- Return indices instead of pointers

```
__bpf_kfunc int bpf_strchr(const char *s__ign, char c)
```

  - Input pointers are unsafe → output pointers would also be unsafe
  - Memory may not be there on successive reads → would still need explicit bounds check

API details

Comparison with stdlib functions

- Return indices instead of pointers

```
__bpf_kfunc int bpf_strchr(const char *s__ign, char c)
```

  - Input pointers are unsafe → output pointers would also be unsafe
  - Memory may not be there on successive reads → would still need explicit bounds check

- Return negative error codes on errors
  - -EFAULT when reading paged out memory
  - -E2BIG when the strings are too big
  - -ERANGE when the strings are outside of the kernel address space
  - -ENOENT as “item not found”

Practical usage

Basic example

Manual implementation

```
SEC("lsm/bprm_check_security")
int BPF_PROG(path_check, struct linux_binprm *bprm) {
    char path[32];
    int path_len, last_slash = -1;

    path_len = bpf_probe_read_str(path, sizeof(path),
                                  bprm->filename);
    if (path_len < 0  path_len >= 32)
        return 0;

    for (int i = 0; i < path_len; i++) {
        if (path[i] == '/')
            last_slash = i;
    }

    // Basename is at path + last_slash + 1
}
```

Using string kfuncs

```
SEC("lsm/bprm_check_security")
int BPF_PROG(path_check, struct linux_binprm *bprm) {
    int last_slash;

    last_slash = bpf_strrchr(bprm->filename, '/');
    if (last_slash < 0)
        return 0;

    // Basename is at bprm->filename + last_slash + 1
}
```

Practical usage

bpftrace extensions

- Thanks to string kfuncs, bpftrace now has more built-in functions for string manipulation:
  - `bool` **`strcontains`**`(string haystack, string needle)`
  - `int64` **`strstr`**`(string haystack, string needle)`
  - `uint64` **`strlen`**`(string exp)`
  - And many more are coming!
- This allows complex in-kernel filtering and string processing directly in bpftrace scripts.

Future directions

- Adding more read-only functions – already happening
- Adding functions which manipulate memory (e.g. `bpf_strncpy`)
  - Problem: source and destination cannot overlap (could be enforced by the verifier?)
  - Is there a use-case for such functions?

- From bpftrace perspective:
  - Avoid `bpf_probe_read_str` when not necessary
  - Add wrappers for more functions

Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.


linkedin.com/company/red-hat


youtube.com/user/RedHatVideos


facebook.com/redhatinc


twitter.com/RedHat

Red Hat